PATH-PLANNING ALGORITHMS IN HIGH-DIMENSIONAL SPACES

A Dissertation
Presented to
The Academic Faculty

By

Florian Hauer

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
Guggenheim School of Aerospace Engineering

Georgia Institute of Technology

May 2019

PATH-PLANNING ALGORITHMS IN HIGH-DIMENSIONAL SPACES

Approved by:

Dr. Tsiotras, Advisor
Guggenheim School of Aerospace Engineering
Georgia Institute of Technology

Dr. Feron
Guggenheim School of Aerospace Engineering
Georgia Institute of Technology

Dr. Vamvoudakis
Guggenheim School of Aerospace Engineering
Georgia Institute of Technology

Dr. Vela
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Boots
School of Interactive Computing
Georgia Institute of Technology

Date Approved: January 7, 2019

A goal without a plan is just a wish.

ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to my advisor, Dr. Panagiotis Tsiotras, for his continuous guidance and support during my doctoral program. Panos gave me the chance to explore many different topics of research while I also worked on applications, broadening both my knowledge and my practical skills. I appreciate the freedom that Panos allowed me in my research, and his knowledgeable feedback about any topic I chose to explore. Panos was very helpful when it came to publishing our findings, working through countless iterations of a document with rigor and patience, until the best formulation was found.

Besides my advisor, I would like to thank Dr. Eric Feron for his help through my entire time at Georgia Tech. Eric helped me find Panos as an advisor before I even arrived in Atlanta and always made sure that everything was going well for me.

I wish to thank the members of my committee, Dr. Kyriakos Vamvoudakis, Dr. Patricio Vela and Dr. Byron Boots, for their insightful comments that led to interesting discussions.

I wish to thank my adoptive labmates, Aude, Manu, Romain and Tim for the numerous brain stimulating coffee breaks.

I wish to thank my parents for their unconditional support, even after years and years of "I'm almost done with school".

Finally, I thank with love my wife, Laura. She has been incredibly supportive and encouraging despite my slow progress. Thank you for taking care of literally everything while I was finishing this thesis during the past few months.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

SUMMARY

Path-planning covers a wide range of applications, from finding the trajectory of an agent in a video game, to understanding how protein folds, to driving autonomous robots in a warehouse, or finding a trajectory that avoids cars and pedestrians for a self-driving car. Three main characteristics are often looked for in the solution of a path-planning algorithm: (1) the solution should avoid collisions with obstacles, (2) the solution should be dynamically feasible, and (3) the solution should be optimal (with the optimality criterion depending on the application: shortest length, fastest time, lowest energy consumption, passenger comfort, passenger safety,...).

Many formulations exist depending on the type of system involved. One major distinction is between systems acting on a continuous search space versus systems acting on a discrete search space. In both cases, the size of the search space affects how fast a solution can be found. For continuous spaces, the dimensionality of the space is a major variable in how fast the solution converges. For discrete spaces, the cardinality of the space is of prime importance: the worst-case runtime increases linearly with the cardinality of the space, in the best case, or even faster, in the general case.

In this thesis, we propose novel algorithms that take advantage of a multi-scale data structure representation of the search-space in order to accelerate path-planning on discrete search spaces. We also make use of more conventional optimization techniques within sampling-based path-planning algorithms to increase the convergence rate of these algorithms.

The main contributions of this thesis are:

1. A path-planning algorithm (the MSPP algorithm) that exploits multi-scale information in any dimension n. This extends previous formulations, which were limited to 2D problems, to any dimension. The algorithm is proven to be complete and the underlying multi-scale data structure used by the algorithm is shown to work directly

with perception algorithms for real-time applications. A theoretical analysis demonstrates the reduction of the complexity from exponential to linear.

2. A probabilistic implementation of the MSPP algorithm (the MSPP-S algorithm). This variant allows the use of the MSPP algorithm without an a priori multi-scale data structure. Sampling is used to estimate the obstacle probabilities, and bounds on the probability of losing completeness are derived. Removing the need to build the multi-scale data structure reduces the runtime of the algorithm by multiple orders of magnitude.
3. A numerical experiment to exhibit convergence properties of sampling-based planning algorithms (the Hypercube Diagonal Experiment). This experiment shows the convergence limits of these algorithms as a function of the dimension of the search space, and matches the theoretical analysis: the number of samples required for convergence increases exponentially with the dimension of the search space.
4. An optimization setup to reduce the error of the value function by repositioning the samples in the search space. The optimization is shown to be easily computable, specifically, the gradient for a specific sample only requires local information. The optimization exhibits good results, in particular, it recovers the visibility graph for a shortest path with polygonal obstacles.
5. A sampling-based algorithm (the DRRT algorithm) that integrates the optimization of the error of the value function within the framework of Rapidly-exploring Random Trees. The DRRT algorithm keeps the quick exploration of the search space from the RRT family in order to find a first solution rapidly, while the added optimization step improves the convergence rate of the algorithm.

CHAPTER 1
INTRODUCTION AND LITERATURE REVIEW

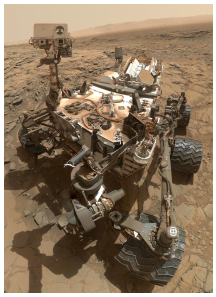
1.1 Introduction

The problem of path-planning consists of finding a trajectory for a system to go from one state to another. A specific example would be a GPS navigation device: it takes a destination as an input, figures out the best route to take the user to his destination, and then gives detailed directions to the user to follow that route. Finding the best route is the role of the path-planning algorithm. In general, path-planning applies to any system which is required to move from a state A to a state B.

The uses of path-planning are increasingly present in our lives with the advances of autonomous technologies. Most autonomous systems utilize path-planning algorithms, including the following:

- The rovers sent to explore Mars, such as Curiosity (see Figure 1.1a), use a set of cameras to build a representation of their surroundings, and then use this representation to find the best path to go to their next destination. For this application, considerations for the best path include energy usage and smoothness of the terrain [1].
- The Roomba vacuum cleaners from iRobot (see Figure 1.1c) use path-planning for two different tasks: planning how to cover the surface to vacuum, and planning how to get back to the base when the battery gets low or the dirt tank is full [2].
- The autonomous driving cars, such as Waymo (see Figure 1.1b), use path-planning at two levels: for navigation to find the best route to follow, and for local planning to assess the best way to drive down a lane while keeping the proper safety distance from other vehicles and pedestrians [3].

Other uses of path-planning include animation and behavior of animated characters for



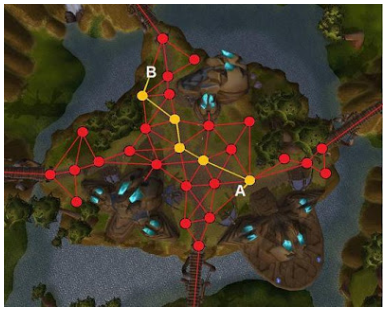
(a) Curiosity [4].

(b) Waymo Car [5].

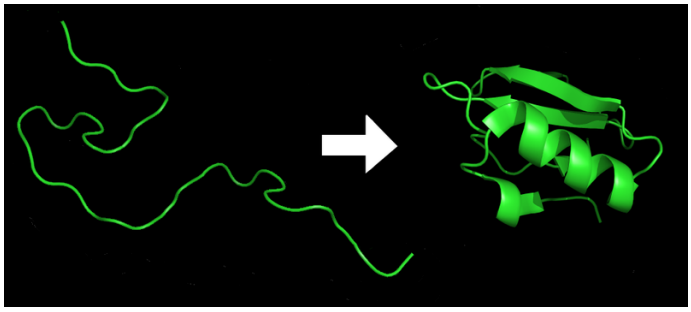
(c) iRobot Roomba [6].

Figure 1.1: Autonomous systems using path-planning algorithms.

movies or video games ([7], [8], [9]), or understanding the complex behavior of complex biologic phenomena such as protein folding. In protein folding ([10],[11]), a high dimensional system (the protein chain) changes its 3D shape from a random initial configuration to a predefined 3D functional shape under the effect of forces between the molecules themselves and the interaction with their environment.



(a) Video game character path-planning [12].



(b) Protein folding [13].

Figure 1.2: Other uses of path-planning algorithms.

Due to the wide use of path-planning applications, many solutions have been developed in order to solve these problems, and many formulations for their solution exist in the literature. These different formulations depend on the type of system, type of environment, and the goal or optimization objective. With GPS navigation, for example, the problem is to find at which intersection to turn from one street to another. The problem can be represented by a graph with nodes corresponding to intersections, and edges corresponding to the streets connecting each intersection ([14], [15], [16]). For the problem of driving

down a lane in traffic, the resolution needed for the solution path is much smaller, and by nature, the problem is continuous. This problem has continuous [17], discrete ([18],[19]) and hybrid formulations [20].

Despite the various formulations, all these algorithms suffer from the same drawback, namely, the size of the search space. This is known as the *curse of dimensionality*. In this thesis, we investigate formulations and algorithms that help reduce the effect of the curse of dimensionality. In particular, we explore the use of multi-scale data structures in order to accelerate the execution of discrete path-planning problems, and also investigate the use of classical optimization techniques, to obtain a faster convergence for sampling-based algorithms.

1.2 Terminology

Let $\mathcal{S}$ be a deterministic agent that moves in a space $\mathcal{W}$. Let π be a trajectory in $\mathcal{W}$, defined as a set of waypoints $\pi_i \in \mathcal{W}$,

$$\pi = [\pi_0, \pi_1, \ldots, \pi_N], \text{ for some } N > 0.$$

Let Π be the set of all trajectories in $\mathcal{W}$. The trajectory π is *feasible* for the agent $\mathcal{S}$ if for each element π_i with $i < N$, the following element π_{i+1} belongs to the reachable set of $\mathcal{S}$ starting at π_i. A cost $J(\pi)$ is associated to each trajectory π. We assume that the cost of a trajectory is the sum of the costs of going from one waypoint to the next,

$$J(\pi) = \sum_{i=0}^{N-1} c(\pi_i, \pi_{i+1}), \tag{1.1}$$

where $c(\pi_i, \pi_{i+1})$ is the cost of going from π_i to π_{i+1} if π_{i+1} belongs to the reachable set of π_i, otherwise $c(\pi_i, \pi_{i+1}) = \infty$.

The notion of obstacles defines forbidden regions of $\mathcal{W}$, leading to potentially infeasible

trajectories. We will assume, for now, that it can be readily checked whether a trajectory is obstacle-free or not. The notion of obstacles will be refined later, once the problem formulation has been made more precise.

The path-planning problem is the following: Given two elements $\mathrm{start}, \mathrm{goal} \in \mathcal{W}$, find an obstacle-free feasible trajectory path π^* such that $\pi^*_0 = \mathrm{start}$, $\pi^*_N = \mathrm{goal}$, and such that π^* minimizes the cost J over all possible trajectories,

$$\pi^* = \underset{\substack{\pi \in \Pi \\ \pi_0 = \mathrm{start} \\ \pi_N = \mathrm{goal} \\ \pi \text{ is feasible and obstacle-free}}}{\arg\min} J(\pi). \tag{1.2}$$

The next section will cover in more detail three formulations of the problem: path-planning on discrete environments, path-planning on continuous environments with sampling-based algorithms, and planning on continuous environments as a parameter optimization problem.

1.3 Literature Review

1.3.1 Discrete Search Spaces

Without loss of generality, we assume that the world information is encoded in a graph $\mathcal{G}$, composed of a set of vertices $\mathcal{V} \subset \mathcal{W}$, and a set of edges $\mathcal{E}$ representing feasible trajectories between the elements of $\mathcal{V}$.

We define the set of *neighbors* $\mathcal{N}(v)$ of an element $v \in \mathcal{V}$ to be the set of elements of $\mathcal{V}$ that are connected to v by an edge in $\mathcal{E}$,

$$\mathcal{N}(v) = \{u \in \mathcal{V} | (u, v) \in \mathcal{E}\}. \tag{1.3}$$

In this thesis, we assume bidirectional edges in $\mathcal{E}$, therefore if v is a neighbor of u then u is a neighbor of v

$$u \in \mathcal{N}(v) \Leftrightarrow v \in \mathcal{N}(u), \quad \forall u, v \in \mathcal{V}. \tag{1.4}$$

The Bellman Operator

Let $J : \mathcal{W} \to \mathbb{R}$, be the *cost-to-go* function. That is, $J(v)$ for $v \in \mathcal{V}$ represents the cost to go from start to v given the current policy. By the term "current policy", we denote the rule that describes the current best trajectory from start to any other node $v \in \mathcal{V}$. An initial guess for J is $J(\mathrm{start}) = 0$ and $J(v) = \infty$, for all $v \in \mathcal{V}$, $v \neq \mathrm{start}$.

Reference [21] introduces the Bellman operator T. This operator applies to a cost-to-go function and returns a new cost-to-go function as follows

$$T(J)(v) = \min_{u \in \mathcal{N}(v)} c(u, v) + J(u). \tag{1.5}$$

The Bellman operator is a contraction [21], therefore it has a unique fixed point J^*, that is,

$$T(J^*) = J^*. \tag{1.6}$$

This fixed point J^* is the minimum cost-to-go for any vertices in $\mathcal{V}$.

It can also be shown that the sequence $J_{i+1} = T(J_i)$ converges within $|\mathcal{V}|$ iterations. Since there are $|\mathcal{V}|$ vertices to evaluate when applying T to J, the solution can be found in $|\mathcal{V}|^2$ iterations.

Once the optimal cost-to-go is found, reconstructing the solution is done following the policy backwards from goal to start using the relation

$$parent(v) = \arg\min_{u \in \mathcal{N}(v)} c(u, v) + J(u), \tag{1.7}$$

where $parent(v)$ defines the unique node preceding v in the optimal trajectory.

The A* Algorithm

The A* algorithm [22] also solves the fixed point of the cost-to-go function. The idea of the A* algorithm is to only apply the Bellman operator to a single node $v \in \mathcal{V}$ per iteration. It

is indeed possible to identify an ordering to process the node of $\mathcal{V}$ such that after processing each of them exactly once, the cost-to-go function will have reached the fixed point of the Bellman operator T.

Algorithm 1: The A* Algorithm

```
CLOSED ← ∅;
OPEN ← {start};
J(v) = ∞, ∀v ∈ V;
J(start) = 0;
while OPEN ≠ ∅ do
    v ← arg min_{u∈OPEN} J(u) + h(u);
    if v == goal then
        return solution;
    OPEN ← OPEN \{v};
    CLOSED ← CLOSED ∪{v};
    foreach u ∈ N(v) do
        if u ∈ CLOSED then
            continue;
        OPEN ← OPEN ∪{u};
        if J(v) + c(v, u) < J(u) then
            J(u) ← J(v) + c(v, u);
```

The A* algorithm keeps a list *OPEN* of the candidate vertices to update at the next iterations, and the list is ordered by cost-to-go plus a heuristic. It is proven that the first element of the list has reached the fixed-point of the Bellman equation, and it can then be removed from that list. It is added to the *CLOSED* list, containing all the vertices already at the fixed-point. All the neighbors of the current element are then updated and added to the *OPEN* list, if needed. The process is repeated until the goal is reached, that is, a solution is found, or until the *OPEN* list is empty, in which case the problem does not have a solution. The A* is proven to have found the optimal path to the goal when the goal is the first element of the *OPEN* list, so no further work is needed when it happens, and the algorithm stops and returns the solution.

The A* algorithm uses a heuristic h in order to direct the search towards the goal. This heuristic furthermore reduces the computation required by only solving the cost-to-

go for the minimum number of vertices needed to reach the goal. Figure 1.3 shows the difference in the space explored with and without the use of a heuristic. To guarantee that the A* algorithm finds the optimal solution, the heuristic needs to be admissible, that is, the heuristic should not over-estimate the actual cost to go from a node v to the goal.

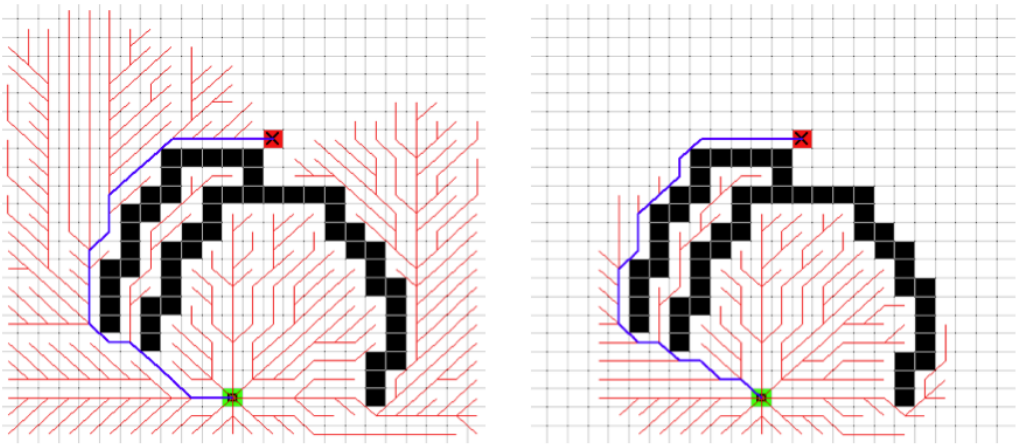


Figure 1.3: Example policies found when reaching the goal by the A* algorithm, $h = 0$ on the left and $h = \|v - \texttt{goal}\|$ on the right. Image from [23].

Re-planning

In the case where the map is updated as the robot moves towards the goal, some algorithms try to reuse the information that has been already computed, rather than redoing all computations from scratch. The D* algorithm [24], including its variants Focussed D* [25] and D*-Lite [26], and the Lifelong Planning A* (LPA*) [27] all reuse the cost-to-go previously computed, to identify regions where the path is not obstacle-free or where the cost could be lowered by checking if the Bellman operator is at its fixed point. The D* algorithm and its variants build the solution backwards, from the goal to the start. This reduces the expected re-planning work to be done as the robot gets closer to the goal, and increases the re-usability of the previously found policy.

When replanning is necessary, some algorithms first focus on finding a feasible solution, and then work on optimizing that solution if time allows. This is the approach used in

the Anytime Dynamic A* [28] and the Anytime D* [29] algorithms. In the case of anytime algorithms, the value function is first populated with suboptimal values corresponding to the first path found, usually using an over-inflated heuristic to quickly find the goal. Once the first solution is found, the value function is then computed with a sub-optimality tolerance. This update involves cost propagation, which can be truncated for unpromising branches without compromising the sub-optimality guarantees of the algorithm. The Truncated LPA* [30] and Anytime Truncated D* [31] algorithms use this idea of truncation in order to accelerate execution.

Reducing the cost by going off the grid

The Field D* [32] and the Theta* [33] algorithms allow for connections that are not defined in the original graph. For example, if 4-connectivity is defined on a 2D grid, that is, the possible movements are going up, down, right or left, then diagonal paths of any angle can be used in order to reduce the cost of the solution, while not increasing the size of the search graph. Interpolation is used to compute the cost of trajectories that are not in the initial search graph. Reference [34] extends this category of algorithms with the Lazy Theta* algorithm, which reorders the expensive operations, in particular, line-of-sight checks, to only be executed if necessary, thus accelerates the execution of the algorithm.

Reducing the size of the search space

The methods described previously, if used only at the finest resolution, become exponentially slow as the dimension increases or as the resolution gets finer. To see this, assume a uniform grid in n dimension is used. To increase resolution, assume each cell is divided by two along each dimension. Starting from a unique cell, and doing k increases in resolution, the size of the search space becomes

$$|\mathcal{V}| = 2^{kn}. \tag{1.8}$$

The variants of both the A* and D* algorithms have a complexity of order $|\mathcal{V}| \log |\mathcal{V}|$ in order to find the optimal solution, so the complexity of solving a path-planning problem on a uniform grid increases exponentially with both k and n.

In order to reduce the size of the search space, representations other than a uniform grid can be chosen. Figure 1.4 shows three representations of the same environment, the first map uses a uniform grid and results in 285 obstacle-free vertices. Large regions of space without obstacles and with uniform cost can be regrouped as a single vertex for the search graph. The second representation does this using a quadtree structure [35], therefore the same environment can be represented with 57 obstacle-free vertices while keeping the same information about the environment. The last representation uses sampling to extract points of the environment that are obstacle-free. This representation has the benefit of having any desired size (the number of samples is chosen), but there is no guarantee about the accuracy of the representation. For example, if there are narrow passages that are unlikely to be sampled, the graph consisting of the samples may have a different topology than the original map.

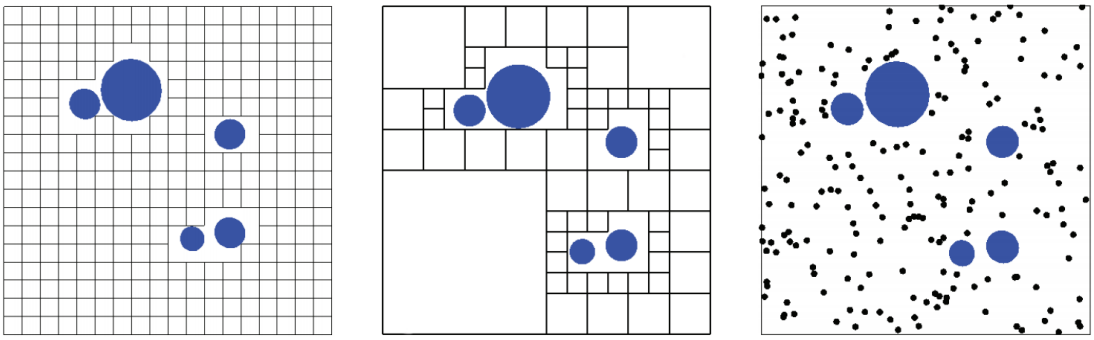


Figure 1.4: Space decomposition using uniform grid (285 obstacle-free cells), quadtree (57 cells) or random sampling (200 points sampled). Image from [36].

A similar idea to reduce the search space is the one of visibility graphs [37]. For path-length optimization, in an environment with polyhedral obstacles, the optimal path passes through the corners of the obstacles, so it suffices to use the corners of the obstacles as the

vertices of the search graph in order to find the optimal solution. Although this formulation can provide extremely fast solutions, it does not extend to more general forms of the cost function.

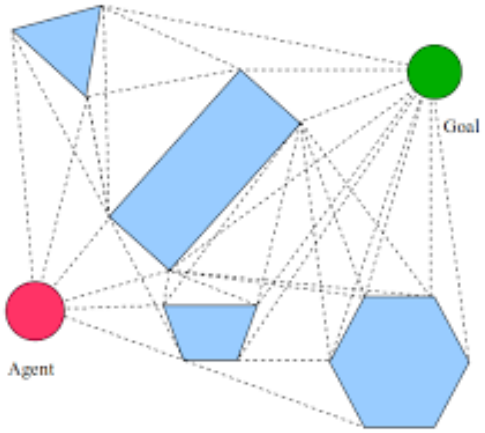


Figure 1.5: Visibility graph example.

Using quadtrees, the search graph contains multiple resolutions since parts of the environment are represented by large areas, while other regions are represented by very small cells. This approach is used in [38], which also decreases the resolution far from the agent in order to further reduce the size of the search space. However, each region of space is only considered at one specific resolution.

Using multi-scale information

Some algorithms go further and use information about a specific region of space at different resolutions during their execution. The idea to organize the information in a multi-resolution manner is motivated by the following two observations: first, processing all information at the finest resolution may be computationally prohibitive, especially for small robotic platforms with limited on-board computational resources. In addition, processing all collected information at the same temporal and spatial scale may not even be necessary for path-planning purposes, as path feasibility is primarily determined by the obstacles in the vicinity of the vehicle. Far away obstacles, on the other hand, affect longer-term objec-

tives, such as exploring the environment, reaching the goal state, etc. Second, a multi-scale hierarchy is often brought about by the perception system itself. Indeed, depending on the sensors, the collected information about the environment is rarely uniformly accurately known, and collected information is often represented by probability values modeling measurement uncertainty [39].

Different approaches have been studied for path-planning using multi-scale maps. Top-down approaches consist of finding a path at the coarser resolution level and progressively enhancing its resolution [40], [41]. These approaches do not guarantee the existence of a path when refining the solution and are thus not complete.

Bottom-up approaches solve the problem at each node at the lowest resolution level and combine the results at different resolution levels. This approach can provide optimality, but requires knowing and processing the entire map at the finest resolution [42], which is very expensive in time and memory to store all the intermediate results.

In [43], it is shown that using bottom-up preprocessing and top-down exploration can lead to finding solutions in sub-linear complexity. This allows the algorithm to find the solutions very fast, but only feasibility can be achieved with this solution, not optimality.

Reference [44] approaches the problem with graphs representing the environment at two scales. The first graph contains the information about the environment at the finest resolution, while the other graph, the coarse graph, contains only a subset of the nodes, uniformly spread over the search space. The algorithm plans on the coarse graph, using another planning routine on subsets of the fine graph in order to evaluate feasibility and cost of the edges. The approach is not optimal but it is proven to be complete [44]. In the worst case, if no solution exists, the fine planning routine ends up running on the full fine resolution graph.

Another approach is to use information at different resolutions at the same time. This idea is explored in [45] by keeping accurate information around the current position of the vehicle and coarser information farther away. The approach is shown to be complete and

very fast. It chooses which resolution to use for each region of space as a function of the position of the agent. 2D Haar wavelets are used in [45] to transform the original map to a multi-resolution map. This algorithm simplifies the environment around the agent location using a wavelet transformation, finds the best path on the simplified environment, moves and then repeats the same operations at the new location. A backtracking scheme prevents the algorithm from getting stuck in an infinite loop, and guarantees finding a solution if one exists.

1.3.2 Continuous Search Space and Sampling-Based Algorithms

Most systems must be discretized in order to perform the search, but the resolution required for an accurate discretization often leads to extremely large search spaces and the use of graph-based algorithms becomes very expensive both in terms of time and memory resources. New algorithms have thus been developed to search the continuous search spaces directly; in particular, sampling-based methods. These methods search for a solution in a graph for which the vertices were sampled from a distribution over the search space $\mathcal{W}$, and the edges were created to connect "nearby" vertices if the trajectory from one vertex to the other is obstacle-free.

There are two main approaches in sampling-based algorithms. The first one consists of sampling a fixed number of points in the search space and using those points to build a graph. The graph is then used in a subsequent step to solve the path-planning problem between any two points in the search space. This approach is used by Probabilistic Road Maps [46] and Fast Marching Trees (FMT* [47]). Some algorithms ([48], [49]) delay the collision checking until an edge is actually needed during the planning phase in order to reduce the initial computation. The second approach does not build the underlying graph, but directly builds the policy as a tree that grows from the initial agent location. This approach, containing the family of Rapidly-exploring Random Trees, is the one we will focus on in this thesis.

The RRT algorithm [50] starts with a single vertex located at the initial position of the agent. At each iteration, the RRT algorithm samples a new vertex, the existing neighboring vertices are found, and the neighbor producing the lower cost is chosen as a parent for the vertex. The structure is a tree that represents the policy found by the algorithm to go from the initial node to any other vertex. The bidirectional RRT algorithm [51] grows two trees, one from the starting node, and one from the goal node. In general, this allows the algorithm to find a solution faster. In [52], extensions of the RRT algorithm are proposed for both hybrid search spaces and control problems. The closed-loop RRT algorithm [53] allows integration of the dynamics of the system and the controllers that will be used to follow the optimal path, and thus the path used for computing the cost and for obstacle checking is the actual path that will be followed by the vehicle.

These variants of the RRT algorithm provide a way to quickly find a solution, but they do not find an optimal solution. Indeed, [54] shows that, with probability one, the RRT algorithm does not return an optimal solution. Reference [54] also introduces the RRT* algorithm, which is very similar to the original RRT algorithm, except that it adds a rewiring step. After a new node is connected to the tree, the Bellman operator is applied to the neighboring nodes in order to minimize their cost using the new sample. This extra step allows the algorithm to converge to the optimal solution as the number of iterations grows towards infinity.

Although the RRT* algorithm converges to the optimal solution, the policy it finds is not optimal with respect to the vertices sampled. The RRT$^\#$ algorithm [55] does the rewiring not only for the neighbors, but keeps propagating the changes down the tree as long as the Bellman operator has not reached the fixed point. The tree found by the RRT$^\#$ is then optimal with respect to the vertices sampled. This provides significant improvements in terms of convergence towards the optimal solution.

RRT^X [56] adds some replanning ideas to the RTT$^\#$ algorithm in order to reuse previous information once the robot moves and new information about the environment is perceived.

It also adds a threshold on the propagation of the rewiring down the tree, and only performs that operation if the expected gain is larger than a predefined value.

The BIT* algorithm [57] uses the same algorithmic core as the RRT$^\#$ algorithm, but processes the samples by batches rather than one by one in order to save computation time.

In [58], the idea to use a gradient descent to move the new sampled point closer to the goal is introduced in order to accelerate finding an initial solution. The idea of moving the new sample is also used in [59] in order to take into account a known drift that applies to the system, such as a river current.

In order to be applicable to control systems, several variants of the RRT* algorithm have been introduced. The LQR-RRT* algorithm [60] generates a local linearization of the system in order to compute the optimal control input to connect two samples. The Kinodynamic RRT* [61] solves a similar problem for linear systems but with a fixed final time constraint when solving the local optimal control problem.

The RRT$^\#$ algorithm will be considered as the base for the future developments in this thesis. It is a good representative of the state-of-the-art in terms of performance, while maintaining a relatively simple formulation. Furthermore, BIT* and RRT^X can be seen as extensions of RRT$^\#$: they add batch processing or relaxation in order to accelerate the execution. The ideas developed in this thesis can be directly integrated into any of these algorithms since they apply to the computational core that they all share.

Below we offer a brief description of the execution of the RRT$^\#$ algorithm. The algorithm is initialized with a single vertex at the initial location of the agent. Until a stopping criterion is reached, the following steps are executed:

- A new sample x_{sample} is drawn and connected to the closest point of the current tree.
- x_{sample} is added to a priority queue, similarly to the A* algorithm.
- While the queue is non-empty, the Bellman operator is applied to the first element of the priority queue, with the k nearest neighbors. Any inconsistent neighbor, in the sense that it does not verify the Bellman equation, is added to the queue.

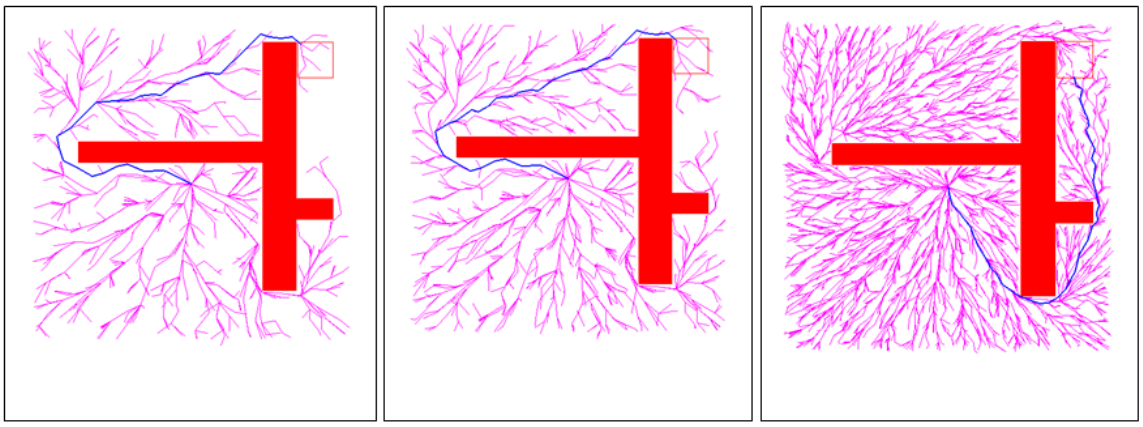


Figure 1.6: Evolution of the RRT* algorithm at 800, 1 200 and 4 200 vertices.

These sampling-based methods suffer less from increasing dimension since the number of sampled points, that is, the size of the graph on which the search is done, can be chosen. A small graph allows fast searches, but convergence to the optimal solution requires the number of samples to go to infinity. That is, the algorithm will most likely find an initial solution fast, but its convergence might be very slow.

Several techniques have been proposed to alleviate the problem of slow convergence. Post-processing [62] of the solution is a common technique, but it only allows for local optimization of the current solution. In order to converge to the optimal solution, the algorithm needs to always return a path near the optimal solution and in the same homotopy class, which cannot be guaranteed in general.

In order to reduce the size of the data structure and maintain a fast execution of the algorithm, the use of a heuristic allows to filter out new samples if they are not promising. Specifically, given an admissible heuristic $h(x_1, x_2)$, that is, a lower-bound on the cost to go from $x_1 \in \mathcal{S}$ to $x_2 \in \mathcal{S}$, and assuming a solution from $x_{\texttt{start}}$ to $x_{\texttt{goal}}$ has cost J, the *relevant region* is defined by

$$\mathcal{X}_{\text{rel}}(J) = \{x | h(x_{\texttt{start}}, x) + h(x, x_{\texttt{goal}}) \leq J\}. \tag{1.9}$$

It is well known that the optimal solution will lie in $\mathcal{X}_{\text{rel}}(J)$, thus samples outside this region

will not be near the optimal solution and will not help convergence. Two methods have been proposed to utilize this information: a) Rejection sampling [63], which samples points in $\mathcal{S}$ but only keeps the points in $\mathcal{X}_{\text{rel}}(J)$; b) Informed sampling [64] samples directly from $\mathcal{X}_{\text{rel}}(J)$, and thus generates only good new samples, but it only works with specific forms of the heuristic, such as Euclidean distance. In addition, for these methods to work well, a good heuristic is required, and finding the best heuristic is often equivalent to solving the original problem.

1.3.3 Path-Planning as an Optimization Problem

Optimal path-planning can also be seen as an optimization problem. Indeed, if a solution is parametrized by x, an optimizer can be used to find the value x^* that optimizes the cost while respecting the constraints of the problem.

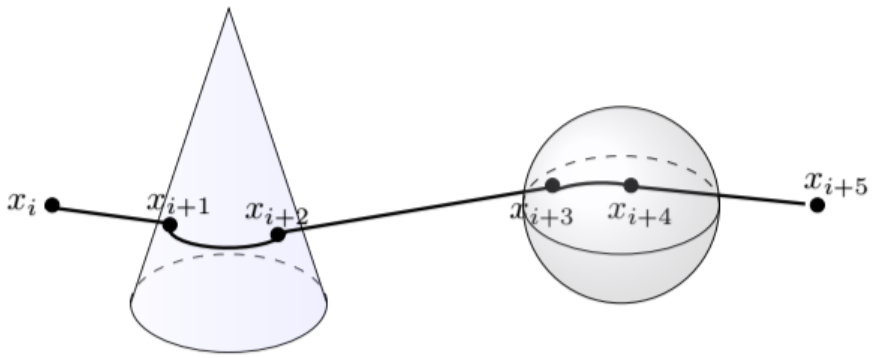


Figure 1.7: Path-planning as an optimization problem (image from [65]).

In [65] and [66], gradient descent and intermittent diffusion were used to solve optimal path-planning problems. Distance functions are used to define obstacles, and the algorithm relies on the fact that the optimal solution is composed of straight lines in free space and geodesics on obstacle boundaries. The problem can then be parametrized by the extremities of each segment of the path. Gradient descent brings the path to a local solution, while intermittent diffusion is used to escape local minima and find the global minimizer.

This approach, however, is limited to geometric shortest path and requires disjoint obstacles with C^2 boundaries. This approach also requires an initial guess, which in general is not easy to find.

Other approaches do not constraint the path to be obstacle-free but rather include the obstacles in the cost function. This idea was used by the CHOMP algorithm [67] and by the STOMP algorithm [68]. Both these approaches are based on a second order gradient optimization to provide a fast convergence to local optima. The STOMP algorithm adds a stochastic step in order to escape local minima and find the global minimizer. These approaches do not scale well with the number of obstacles, they require to build a computationally expensive obstacle distance map over the search space and the number of local minima on this distance map increases exponentially with the number of obstacles. Another problem with these methods is the fact that the solution path is not guaranteed to be obstacle-free since there is no hard constraint for obstacles.

A different family of optimization algorithms was used in [69] and [70], where genetic algorithms and simulated annealing were used in order to optimize paths. These approaches allow multiple optimization objectives, but suffer from the fact that they require a large population of paths to initialize the algorithm.

1.4 Summary

Many different approaches exist to solve path-planning problems.

In the case of grid-based algorithms, the A* algorithm is optimal in the sense that it finds the optimal solution while exploring the minimum number of vertices. However, as the search space grows, the A* algorithm becomes too slow. Multi-scale techniques have been used but they all suffer from either not being complete [41], not being optimal [43], or requiring time-consuming preprocessing of the map [42]. Reference [71] shows an algorithm not suffering from any of these three problems, but it is limited to 2D path-planning problems.

Sampling-based algorithms suffer less from the search space dimensionality increase in the sense that they will, in general, find a solution rather fast, but they suffer from slow convergence when the search space is large. Formulating the problem as a parameter optimization problem allows fast convergence, but the formulation presented in [65] is limited to shortest length paths, specific obstacle formulations, and it requires an initial guess of the solution.

CHAPTER 2

DISCRETE SEARCH SPACES

2.1 Problem Formulation

2.1.1 Multi-Resolution World Representation

The world $\mathcal{W} \subset \mathbb{R}^d$ is represented as a d-dimensional grid. Without loss of generality, we assume that each elementary cell of the grid is a unit hypercube and that the entire world is contained within a hypercube of side length 2^ℓ.

The world $\mathcal{W}$ is not perfectly known, but an abstraction of the world as a tree $\mathcal{T} = (\mathcal{N}, \mathcal{R})$ can be obtained from a set of measurements $\mathcal{D}$. The nodes of $\mathcal{T}$, denoted $\mathcal{N}$, are connected by edges $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{N}$. The nodes encode, in a hierarchically organized structure, all the information collected about the environment. Specifically, the nodes of $\mathcal{T}$ at depth ℓ correspond to the grid cells (unit hypercubes) whereas higher-level nodes correspond to hypercubes of larger side lengths, consisting of unions of hypercubes at lower levels. At depth $\ell = 0$ we have a single node representing the whole search space. The information contained in a given tree node $n \in \mathcal{T}$ is the probability that the corresponding cell $H(n) \subseteq \mathcal{W}$ is an obstacle.

We will keep track of the nodes of $\mathcal{T}$ using two indices, k and p. Specifically, each node $n_{k,p} \in \mathcal{N}$ has the following properties:

- The node represents a hypercube $H(n_{k,p}) \subseteq \mathcal{W}$ of side length 2^k and volume 2^{dk} centered at $p = [p^1, p^2, \ldots, p^d]$, where, for all $i \in [1, d]$, p^i belongs to the set $\{(2j + 1)2^k : -2^{\ell-k} \leq 2j + 1 \leq 2^{\ell-k}\}$, except for $k = \ell$, where the only node $n_{\ell,p}$ is centered at $p = [0, 0 \ldots, 0]$.
- The node is present at depth $\ell - k$ in the tree $\mathcal{T}$.
- The *children* of the node are $n_{k-1,q_i}, i \in [1, 2^d]$ where $q_i = p + 2^{k-2}e_i$ and where

e_i is each of the 2^d (d-dimensional) vectors generated by $[\pm 1, \pm 1, \ldots, \pm 1]$. The hypercubes associated with the children of node $n_{k,p}$ induce a dyadic partitioning of $H(n_{k,p})$, as

$$H(n_{k,p}) = \bigcup_{i=1}^{2^d} H(n_{k-1,q_i}). \tag{2.1}$$

- The node is a *leaf* of $\mathcal{T}$ if it has no children.
- The *value* of a node is denoted by $V(n_{k,p})$. If the node is a leaf node, this value is the probability that the node $n_{k,p}$ is an obstacle. If the node has children, this value is the average of the values of its children. That is,

$$V(n_{k,p}) = \begin{cases} P(o_{n_{k,p}} = 1|\mathcal{D}), & \text{if } n_{k,p} \text{ is a leaf,} \\ \frac{1}{2^d} \sum_{i=1}^{2^d} V(n_{k-1,q_i}), & \text{otherwise,} \end{cases} \tag{2.2}$$

  where $o_{n_{k,p}} \in \{0, 1\}$ is the binary occupancy label associated with $H(n_{k,p})$ and $\mathcal{D}$ is the set of measurements used to generate $\mathcal{T}$. Here, $P(o_{n_{k,p}} = 1|\mathcal{D})$ denotes the probability that $H(n_{k,p})$ is an obstacle given the set of measurements $\mathcal{D}$.

The set of all leaf nodes is a partition of the entire space $\mathcal{W}$ and represents the most accurate information collected about the environment. We say that a leaf node represents the *finest information*, in the sense that it has the most accurate information available at that location.

Figure 2.1 shows an example of the tree structure in 1-D. The horizontal lines around each node in the figure represent the hypercube associated to that node. The hypercube in this case is a line segment.

Note that, for convenience, we will often refer to a node in lieu of the region represented by that node. This slight abuse of terminology should not pose any problems since there is a one-to-one correspondence between nodes in $\mathcal{T}$ and the regions they represent in $\mathcal{W}$.

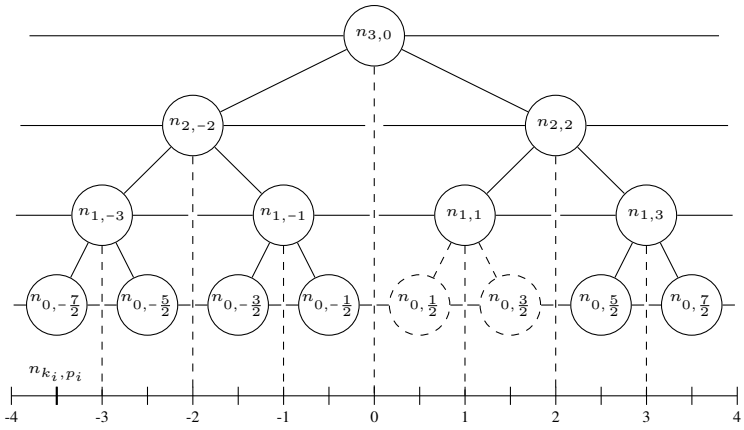


Figure 2.1: 1-D Example of the tree $\mathcal{T}$.

2.1.2 The Path-Planning Problem

Two nodes of $\mathcal{T}$ are *neighbors* if their corresponding hypercubes share a face, specifically, their intersection is a hypercube of dimension $d-1$. A necessary and sufficient condition for two nodes n_{k_1,p_1} and n_{k_2,p_2} to be neighbors is that both of the following two conditions are satisfied:

- The following expression holds:

$$\|p_1 - p_2\|_\infty = 2^{k_1-1} + 2^{k_2-1}, \tag{2.3}$$

- There exists a unique $i \in [1, d]$, such that

$$|(p_1 - p_2)_i| = 2^{k_1-1} + 2^{k_2-1}, \tag{2.4}$$

  where $(p_1 - p_2)_i$ is the i^{th} component of the vector.

In the case of a 2-D or a 3-D environment with a uniform grid, conditions (2.3) and (2.4) imply 4-connectivity or 6-connectivity, respectively.

We define a *path* $\pi = (n_{k_1,p_1}, n_{k_2,p_2}, \ldots, n_{k_N,p_N})$ in $\mathcal{T}$ to be a sequence of nodes $n_{k_i,p_i} \in \mathcal{N}$, each at corresponding position p_i and depth $\ell - k_i$, such that two consecutive nodes of the sequence are neighbors. A path is a *finest information path* (FIP) if all its nodes are leafs

of $\mathcal{T}$. Note that a finest information path may contain nodes that are not of unit size. Due to noisy measurements, and to prevent overconfidence and numerical issues, perception algorithms will not allow $P(o_{n_{k,p}}|\mathcal{D})$ to reach 100%. So we introduce the notion of ε-obstacles. Specifically, given $\varepsilon \in [0, 1)$, a node $n_{k,p} \in \mathcal{N}$ is an *ε-obstacle* if

$$V(n_{k,p}) \geq 1 - 2^{-dk}\varepsilon. \tag{2.5}$$

For a node corresponding to a unit cell ($k = 0$), the threshold becomes $1 - \varepsilon$, so any unit cell with a probability higher than $1 - \varepsilon$ is an ε-obstacle. A path π is *ε-feasible* if none of its nodes are ε-obstacles.

Proposition 1. *If a node in $\mathcal{T}$ is an ε-obstacle, then all leaf nodes descendant from this node are also ε-obstacles.*

Proof. Let $n_{k,p} \in \mathcal{N}$ and let n_{m_i,q_i}, where $i \in [1, L]$ be the descendant leaf nodes of $n_{k,p}$. Using equation (2.2) recursively until we find the descendant leafs of $n_{k,p}$, we obtain that

$$V(n_{k,p}) = \frac{1}{2^{dk}} \sum_{i=1}^{L} 2^{dm_i} V(n_{m_i,q_i}) \tag{2.6}$$

$$= \sum_{i=1}^{L} 2^{d(m_i-k)} V(n_{m_i,q_i}). \tag{2.7}$$

Since the leaf nodes descendent from $n_{k,p}$ form a partition of $H(n_{k,p})$ (see (2.1)), it follows that $2^{dk} = \sum_{i=1}^{L} 2^{dm_i}$ and hence $1 = \sum_{i=1}^{L} 2^{d(m_i-k)}$.

Suppose now that $n_{k,p}$ is a ε-obstacle and suppose, on the contrary, that there exists n_{m_j,q_j} such that n_{m_j,q_j} is not a ε-obstacle, i.e., suppose that $V(n_{m_j,q_j}) < 1 - 2^{-dm_j}\varepsilon$. Since

$n_{k,p}$ is an ε-obstacle, it follows from (2.5) that $V(n_{k,p}) \geq 1 - 2^{-dk}\varepsilon$. Thus, from (2.7),

$$\begin{aligned} V(n_{k,p}) &= \sum_{i=1, i\neq j}^{L} 2^{d(k-m_i)} V(n_{m_i,q_i}) + 2^{d(k-m_j)} V(n_{m_j,q_j}) \\ &< 1 - 2^{d(m_j-k)} + 2^{d(m_j-k)}(1 - 2^{-dm_j}\varepsilon) \\ &< 1 - 2^{-dk}\varepsilon, \end{aligned} \tag{2.8}$$

leading to a contradiction. Hence all descendant leafs of $n_{k,p}$ are ε-obstacles. □

The previous proposition guarantees that, for any given $\varepsilon \in [0, 1)$, if a node at any resolution level is an ε-obstacle, then all its descendants are also ε-obstacles. Using a threshold for the ε-obstacle that depends on the size of the node extends the work in [71] to any value of $\varepsilon \in [0, 1)$ instead of only small enough ε.

Given the representation of $\mathcal{W}$ encoded in the tree $\mathcal{T}$, the problem is to find a ε-feasible FIP between two nodes in the tree, n_{start}, representing the starting node, and n_{goal}, representing the goal node, and to report failure if no such path exists.

2.2 The Multi-Scale Path-Planning (MSPP) Algorithm

The overall idea of the Multi-Scale Path-Planning (MSPP) algorithm is to iteratively solve smaller problems instead of solving the original problem at once. Starting at iteration $i = 0$ from $n_{k_0,p_0} = n_{\text{start}}$, the algorithm uses $\mathcal{T}$ to create a local representation of the environment encoded in a graph $\mathcal{G}_i$, called the *reduced graph*. Briefly, the nodes of $\mathcal{G}_i$ are a collection of nodes of $\mathcal{T}$ forming a partition of the search space, with fine resolution around the current node, say n_{k_i,p_i}, at iteration i, and with progressively coarser resolutions away from n_{k_i,p_i} (see Figure 2.2). The resolution levels and the horizon of each resolution level are controlled by the parameters ℓ and α (see Section 2.3.3). The graph $\mathcal{G}_i$ is thus a spatial representation of $\mathcal{W}$, as opposed to $\mathcal{T}$ which is a hierarchical representation of $\mathcal{W}$. A sequence of path-planning problems are solved in the graphs $\mathcal{G}_i$, ($i = 0, 1, 2, \ldots$)

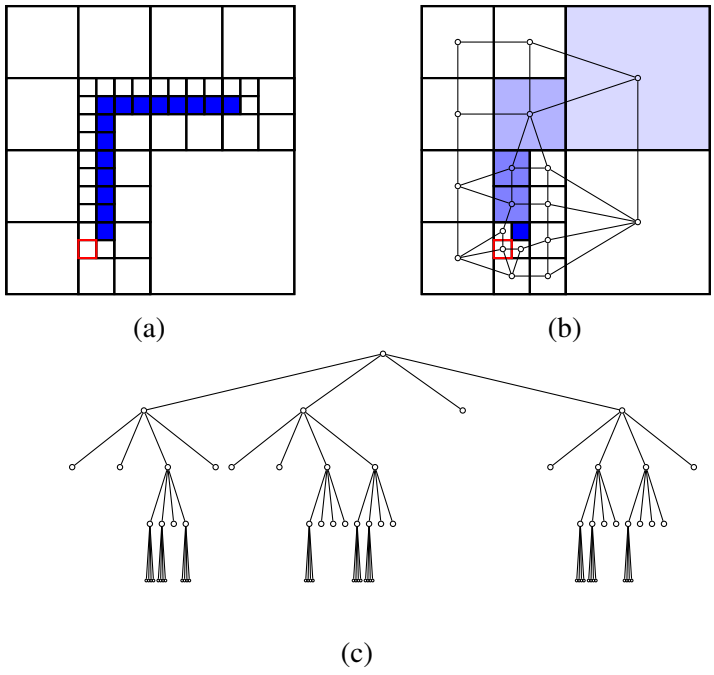


Figure 2.2: (a) Example environment $\mathcal{W}$ with the current cell, n_{k_i,p_i}, shown in red. Obstacles are drawn with solid blue color; (c) Reduced graph $\mathcal{G}_i$ around n_{k_i,p_i} in red and corresponding space partition; (b) Corresponding tree $\mathcal{T}$, children order: top-left, bottom-left, bottom-right, top-right.

as follows. The shortest path $\pi_i^{\text{goal}} = (n_{k_i,p_i}, \pi_1^i, \pi_2^i, \pi_3^i, \ldots, n_{\text{goal},i})$ from n_{k_i,p_i} to $n_{\text{goal},i}$ is found in $\mathcal{G}_i$, where $n_{\text{goal},i}$ is the unique node of $\mathcal{G}_i$ containing n_{goal}. Then $n_{k_{i+1},p_{i+1}}$ is set to the node π_1^i, and the process is repeated for the next iteration until $\pi_1^i = n_{\text{goal},i}$ for some $i \geq 0$. By construction, the nodes of $\mathcal{G}_i$ neighboring n_{k_i,p_i} are necessarily leafs of $\mathcal{T}$ and hence are finest information nodes. It follows that, at termination, the path $\pi = (n_{k_0,p_0}, n_{k_1,p_1}, \ldots, n_{\text{goal}})$ is a FIP in $\mathcal{T}$.

The details of the construction of $\mathcal{G}_i$ are given in Section 2.2.1 below.

2.2.1 Reduced Graph Construction

In this section, we give the details of the construction of the reduced graph $\mathcal{G}_i$ used by the search algorithm at each iteration. To construct the reduced graph $\mathcal{G}_i$, first the vertices of $\mathcal{G}_i$ are selected by traversing the tree $\mathcal{T}$, and then edges are created between vertices representing neighboring nodes. The vertices of $\mathcal{G}_i$ are selected recursively starting from

the root of $\mathcal{T}$ by the function `GetReducedGraphVertices`. A node $n_{k,p} \in \mathcal{N}$ is selected to be included in $\mathcal{G}_i$ if all the following conditions are true:

- The node $n_{k,p}$ is a leaf, or

$$\|p - p_i\|_2 - \frac{\sqrt{d}}{2} 2^{k_i} > \alpha 2^k, \tag{2.9}$$

  where p_i and $\ell - k_i$ are the position and the depth of the current node n_{k_i,p_i}, respectively, and $\alpha > 0$ is a parameter.

- The node $n_{k,p}$ is not an ε-obstacle, as defined in (2.5).
- The node $n_{k,p}$ does not contain a part of the *partial path candidate* $\pi^i_{\text{start}} = (n_{k_0,p_0}, n_{k_1,p_1}, \ldots, n_{k_i,p_i})$.

If a node is not selected, its children are then considered next, and the process repeats itself until all nodes of $\mathcal{T}$ have been examined.

Equation (2.9) provides a condition to select a node depending on how large the node is and how far away it is from n_{k_i,p_i}. The result of this condition is to select unit size nodes in the sphere of radius α centered at p_i, level 2 size nodes in the sphere of radius 2α, ..., level k size nodes in the sphere of radius $2^k\alpha$, etc. The selected nodes thus form a partition of the search space from which we have removed ε-obstacles and all nodes corresponding to the current partial path candidate, since we want a loopless ε-feasible solution. Every pair of nodes selected is tested against the neighborhood tests (2.3) and (2.4), and edges are created for neighboring nodes.

Figure 2.3 shows an example of the construction of the graph $\mathcal{G}_i$ from $\mathcal{T}$ for $\alpha = 1/2$ and $n_{k_i,p_i} = n_{0,-\frac{7}{2}}$ for a simple, 1D problem ($\mathcal{W} = [-2^3, 2^3]$). The selection starts at the root of $\mathcal{T}$, $n_{3,0}$, which is not a leaf and the condition (2.9) is not true. Therefore, the node is not selected and its children are considered next (red edge). The node $n_{2,2}$ passes the selection test and it is added as a vertex in $\mathcal{G}_i$ (red node). After the recursion, $\mathcal{G}_i$ is created with the selected nodes and with edges representing spatial connectivity between

Function GetReducedGraphVertices($n_{k,p}$,$vertices$,n_{k_i,p_i})

```
Function GetReducedGraphVertices (n_{k,p},vertices,n_{k_i,p_i})
    Data: Node n_{k,p},Vertex list vertices, Current node n_{k_i,p_i}
    if (||p − p_i||_2 − (√d/2)2^{k_i} ≥ α2^k OR isLeaf(n_{k,p})) AND
     doesNotContainPath(n_{k,p}) then
        if cost(n_{k,p})< M then
            vertices ← vertices ∪ n_{k,p};
    else
        foreach n_{m,q} child of n_{k,p} do
            GetReducedGraphVertices (n_{m,q},vertices,n_{k_i,p_i});
    return vertices;
```

the selected nodes. The graph $\mathcal{G}_i$ is shown at the bottom of Figure 2.3.

2.2.2 Backtracking Algorithm

The proposed MSPP algorithm is a backtracking algorithm. At each iteration i, the algorithm creates the reduced graph $\mathcal{G}_i$ as detailed in Section 2.2.1. Then a search algorithm to find the shortest path from the current node to the goal node is applied. To understand the details of the implementation, let $visits(n_{k_i,p_i})$ denote the set of neighbors of the current node at iteration i already visited by the algorithm. For $i = 0$, let this set be the empty set. Let $\pi_i^{\text{goal}} = (n_{k_i,p_i}, \pi_1^i, \pi_2^i, \pi_3^i, \ldots, n_{\text{goal},i})$ denote the shortest path from n_{k_i,p_i} to $n_{\text{goal},i}$ on the graph $\tilde{\mathcal{G}}_i = \mathcal{G}_i \setminus visits(n_{k_i,p_i})$. The graph $\tilde{\mathcal{G}}_i$ is constructed from $\mathcal{G}_i$ by removing all nodes in $visits(n_{k_i,p_i})$ along with the corresponding edges. If π_i^{goal} exists then the algorithm *moves forward*, that is, the next node in π_i^{goal} is selected as the current node, $n_{k_{i+1},p_{i+1}} \leftarrow \pi_1^i$, and π_{start}^i is updated as $\pi_{\text{start}}^{i+1} \leftarrow (\pi_{\text{start}}^i, \pi_1^i)$. If π_i^{goal} does not exist, it follows that no path from the current node n_{k_i,p_i} to n_{goal} is ε-feasible and the algorithm backtracks. In this case the set $visits(n_{k_i,p_i})$ is reset to the empty set, the current node is removed from the partial solution candidate, that is, π_{start}^i is updated as $\pi_{\text{start}}^{i+1} \leftarrow \pi_{\text{start}}^i \setminus n_{k_i,p_i}$ and the next node is chosen as the last element of the partial solution, that is, $n_{k_{i+1},p_{i+1}} \leftarrow \text{lastElement}(\pi_{\text{start}}^{i+1})$. Subsequently, the set $visits(n_{k_{i+1},p_{i+1}})$ is updated accordingly from $visits(n_{k_{i+1},p_{i+1}}) \leftarrow visits(n_{k_{i+1},p_{i+1}}) \cup \{n_{k_i,p_i}\}$ and the algorithm pro-

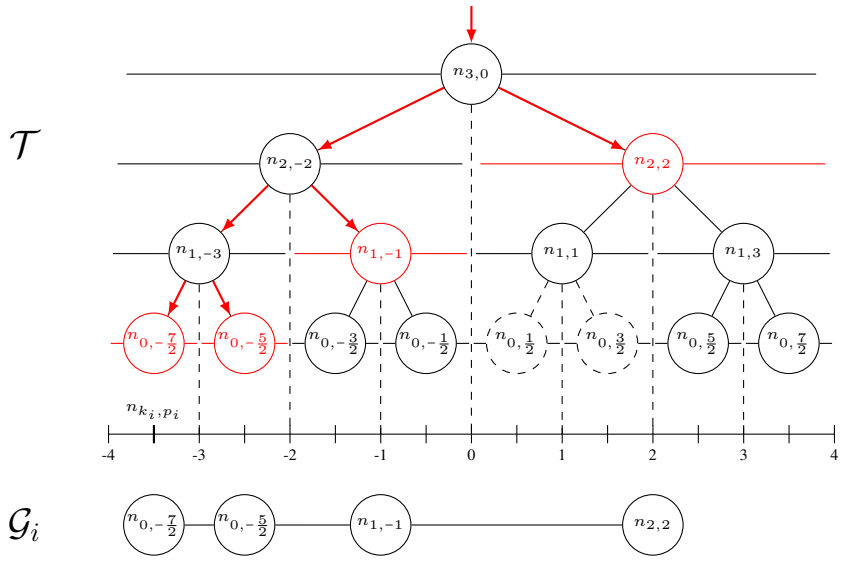


Figure 2.3: 1-D Example of the tree $\mathcal{T}$ and the corresponding graph $\mathcal{G}_i$ for $\alpha = 0.5$ and $n_{k_i,p_i} = n_{0,-\frac{7}{2}}$. The red nodes of $\mathcal{T}$ are selected by the algorithm to be the vertices of $\mathcal{G}_i$ and the red edges represent the calls to the `GetReducedGraphVertices` function.

ceeds to the next iteration.

Once the algorithm reaches the goal, it terminates and returns the solution path stored in π^i_{start}. Note that if the algorithm backtracks when $n_{k_i,p_i} = n_{\text{start}}$ then it will have tried every possible path without finding a solution, in which case it reports failure.

The MSPP algorithm is shown in Algorithm 2. It works as follows:

- [Lines 1–2] The variables are initialized:
  - the iteration number is set to 0, the position of the algorithm n_{k_i,p_i} and the partial candidate π^i_{start} are both initialized to the start node n_{start} (line 1)
  - the set of visited neighbors are initialized empty for all nodes of $\mathcal{T}$ (line 2)
- [Lines 3–19] The main loop is called until a solution is found or failure is detected:
  - $\mathcal{G}_i$ is computed, and the vertices $v_{\text{start},i}$, $v_{\text{goal},i}$ corresponding to n_{k_i,p_i} and n_{goal} are identified (line 4)
  - the shortest path π^{goal}_i from $v_{\text{start},i}$ to $v_{\text{goal},i}$ on $\tilde{\mathcal{G}}_i$ is solved (line 5)
  - if π^{goal}_i exists, the algorithm moves forward (lines 7–9)
  - otherwise, the algorithm backtracks (lines 13–18)
  - and the algorithm goes to the next iteration (line 18)

- The algorithm stops when:
  - the goal is reached (lines 10-11)
  - the algorithm backtracks from n_{start}, that is, it has explored every possible path without success (lines 15-16)

In our implementation the cost of traversing a node is chosen as

$$\text{cost}(n_{k,p}) = \begin{cases} 2^{dk}(\lambda_1 V(n_{k,p}) + \lambda_2), & \text{if } n_{k,p} \text{ is not an } \varepsilon\text{-obstacle,} \\ M, & \text{otherwise,} \end{cases} \tag{2.10}$$

where $V(n_{k,p})$ as in (2.2), and $\lambda_1, \lambda_2 \in (0, 1]$. Other costs can be chosen depending on the problem at hand. The cost in (2.10) takes into account the probability that the node $n_{k,p}$ is an ε-obstacle with weight λ_1 and it adds the length of the path with weight λ_2, scaled by the volume of the hypercube corresponding to the node. The scaling guarantees that any ε-obstacle-free path contained within a node $n_{k,p}$ will have a cost smaller than the cost of that node. In (2.10), the value of M should be chosen large enough to avoid obstacles. In fact, we have the following result.

Proposition 2. *A loopless path is ε-feasible if and only if its cost is less than M, where $M > 2^{d\ell+1}$.*

Proof. Let $\pi = (n_{k_i,p_i})_{i=1}^N$ be an ε-feasible path without loops. It follows

$$\text{cost}(\pi) = \sum_{i=1}^{N} 2^{dk_i}(\lambda_1 V(n_{k_i,p_i}) + \lambda_2) \leq 2 \sum_{i=1}^{N} 2^{dk_i}, \tag{2.11}$$

since $\lambda_1, \lambda_2, V(n_{k,p}) \leq 1$. Since the path is loopless, the volume of all nodes in the path is necessarily smaller that the volume of the whole space, hence $\sum_{i=1}^{N} 2^{dk_i} \leq 2^{d\ell}$. It follows that

$$\text{cost}(\pi) \leq 2 \sum_{i=1}^{N} 2^{dk_i} \leq 2 \times 2^{d\ell} = 2^{d\ell+1} < M. \tag{2.12}$$

Conversely, let $\pi = (n_{k_i,p_i})_{i=1}^{N}$ be a path that is not ε-feasible. Then there exists $i' \in [1, N]$ such that $n_{k_{i'},p_{i'}}$ is an ε-obstacle, so

$$\text{cost}(\pi) = \sum_{i=1}^{N} \text{cost}(n_{k_i,p_i}) \tag{2.13}$$

$$= \sum_{i=1, i\neq i'}^{N} \text{cost}(n_{k_i,p_i}) + \text{cost}(n_{k_{i'},p_{i'}}) \tag{2.14}$$

$$= \sum_{i=1, i\neq i'}^{N} \text{cost}(n_{k_i,p_i}) + M \geq M. \tag{2.15}$$

□

Note that we want to detect non-promising partial path candidates π^i_{start} as soon as possible in order to backtrack early on and avoid spending computational resources completing a partial path candidate that will not lead to a valid solution. The following corollary guarantees the existence of a path in $\tilde{\mathcal{G}}_i$ when there exists a ε-feasible FIP contained in the space represented by $\tilde{\mathcal{G}}_i$, in particular, its contrapositive guarantees that there will be no ε-feasible FIP contained in the space represented by $\tilde{\mathcal{G}}_i$ if there is no path in $\tilde{\mathcal{G}}_i$.

Corollary 2.2.1. *Suppose there exists an ε-feasible FIP from n_{k_i,p_i} to n_{goal} contained in the region of space represented by $\tilde{\mathcal{G}}_i$, then there exists an ε-feasible path in $\tilde{\mathcal{G}}_i$ from n_{k_i,p_i} to $n_{\text{goal},i}$.*

Proof. Suppose there exists an ε-feasible FIP $\pi = (\pi_1, \pi_2, \ldots, \pi_L)$ from n_{k_i,p_i} to n_{goal} contained in the region of space represented by $\tilde{\mathcal{G}}_i$.

Each π_j is contained in a node of $\tilde{\mathcal{G}}_i$, say g_j. The path π is ε-feasible, so π_j is not an ε-obstacle. Also π_j is a descendant of g_j in $\mathcal{T}$, so by the contrapositive of Proposition 1, g_j is not an ε-obstacle. Note that $g_1 = \pi_1 = n_{k_i,p_i}$ and $g_L = n_{\text{goal,i}}$. The path $(g_1, g_2, \ldots, g_L)$ is then an ε-feasible path from n_{k_i,p_i} to $n_{\text{goal},i}$ in $\tilde{\mathcal{G}}_i$. □

Algorithm 2: The MSPP Algorithm

Data: Tree $\mathcal{T}$, Start node n_{start}, Goal node n_{goal}
Result: ε-feasible FIP from n_{start} to n_{goal} or failure

$i \leftarrow 0, n_{k_i,p_i} \leftarrow n_{\text{start}}$,$\pi^0_{\text{start}} \leftarrow [n_{k_i,p_i}]$;
visits($n_{k,p}$)$\leftarrow \emptyset$,$\forall n_{k,p}$;
while `cost`(π^i_{start})$\leq M$ **do**
    $(\mathcal{G}_i, v_{\text{start},i}, v_{\text{goal},i}) \leftarrow$`ReducedGraph`($\mathcal{T}$,$n_{k_i,p_i}$);
    $\pi^{\text{goal}}_i \leftarrow$`SP`($\mathcal{G}_i, v_{\text{start},i}, v_{\text{goal},i}$,visits($n_{k_i,p_i}$));
    if `exists`(π^{goal}_i) **then**
        $n_{k_{i+1},p_{i+1}} \leftarrow$`firstElement`($\pi^{\text{goal}}_i$);
        visits(n_{k_i,p_i})$\leftarrow$visits(n_{k_i,p_i})$\cup n_{k_{i+1},p_{i+1}}$;
        $\pi^{i+1}_{\text{start}} \leftarrow [\pi^i_{\text{start}} \quad n_{k_{i+1},p_{i+1}}]$;
        if $n_{k_{i+1},p_{i+1}}$=n_{goal} **then**
            return π^{i+1}_{start};
    else
        visits(n_{k_i,p_i})$\leftarrow \emptyset$;
        π^{i+1}_{start}=`removeLastElement`(π^i_{start});
        if $\pi^{i+1}_{\text{start}} = \emptyset$ **then**
            Report failure;
        else
            $n_{k_{i+1},p_{i+1}} \leftarrow$`lastElement`($\pi^{i+1}_{\text{start}}$);
    $i \leftarrow i + 1$;
Report failure;

2.3 Algorithm Properties

2.3.1 Completeness

In this section we prove that the MSPP algorithm is complete, that is, it terminates after a finite number of iterations and returns a solution, if one exists or reports failure otherwise.

At iteration i, let a *valid extension* of π^i_{start} be a ε-feasible FIP from n_{k_i,p_i} to n_{goal} that does not intersect with π^i_{start}.

The following proposition guarantees that at iteration i, any valid extension of π^i_{start} is contained in the region of space represented by $\tilde{\mathcal{G}}_i$.

Proposition 3. *Let i be the iteration number.*

Suppose that, for each iteration $j = 0, 1, \ldots, i-1$, the MSPP algorithm backtracked only

if there was no valid extension of π^j_{start}. Then, at iteration i, any valid extension of π^i_{start} is contained in the space represented by $\tilde{\mathcal{G}}_i$.

Proof. Suppose that for each iteration $j = 0, 1, \ldots, i-1$, the MSPP algorithm backtracked only if there was no valid extension of π^i_{start}.

Suppose, by contradiction, that there exists a valid extension π of π^i_{start} that is not fully contained in $\tilde{\mathcal{G}}_i$. Then there exist a node of π, say $n_{k,p}$, that is not contained in $\tilde{\mathcal{G}}_i$. π is a ε-feasible FIP, so $n_{k,p}$ is a leaf of $\mathcal{T}$ and $n_{k,p}$ is not an ε-obstacle and π does not intersect with π^i_{start} , so $H(n_{k,p})$ does not intersect with nodes of π^i_{start}. Those conditions guarantee that $n_{n,p}$ or one of its ancestors is in $\mathcal{G}_i$, so $n_{k,p}$ is contained in the space represented by $\mathcal{G}_i$. Also, by assumption, the node $n_{k,p}$ is not in $\tilde{\mathcal{G}}_i$, so it is in $\mathcal{G}_i \setminus \tilde{\mathcal{G}}_i$, that is, it is a visited neighbor of n_{k_i,p_i}. Then there exists an iteration $j < i$ such that the algorithm backtracked from $n_{k,p}$, that is, there are no valid extensions of π^j_{start}. π is a valid extension of π^i_{start} that passes through $n_{k,p}$, an already visited neighbor of n_{k_i,p_i}. In particular, there exists an ε-feasible FIP from $n_{k,p}$ to n_{goal} that does not intersect with π^i_{start}. This is a contradiction. Thus there is no valid extension of π^i_{start} that is not fully contained in the space represented by $\tilde{\mathcal{G}}_i$. □

Proposition 4. *If the MSPP algorithm backtracks at iteration i, then there is no valid extension of π^i_{start}.*

Proof. Recall that the algorithm backtracks at iteration i if no ε-feasible path from n_{k_i,p_i} to $n_{\text{goal},i}$ exists in $\tilde{\mathcal{G}}_i$.

For iteration $j = 0$, the region of space represented by $\tilde{\mathcal{G}}_0$ is the entire environment $\mathcal{W}$ from which we have removed some ε-obstacles. Suppose there is no ε-feasible path from n_{start} to $n_{\text{goal},0}$ in $\tilde{\mathcal{G}}_0$, then, by the contrapositive of Corollary 2.2.1, there is no ε-feasible FIP from n_{start} to n_{goal} contained in the space represented by $\tilde{\mathcal{G}}_0$, so no ε-feasible FIP from n_{start} to n_{goal} in $\mathcal{W}$.

Assume now that Proposition 4 is true for all $j = 0, 1, \ldots, i-1$ and suppose that the MSPP

algorithm backtracks at iteration i. Hence, there is no ε-feasible path from n_{k_i,p_i} to $n_{\text{goal},i}$ in $\tilde{\mathcal{G}}_i$. On one hand, the contrapositive of Corollary 2.2.1 implies that there is no ε-feasible FIP that is fully contained in $\tilde{\mathcal{G}}_i$. On the other hand, with the induction assumptions, Proposition 3 guarantees that any valid extension of π^i_{start} is contained in the space represented by $\tilde{\mathcal{G}}_i$. Thus, there cannot be a valid extension of π^i_{start}.

□

Proposition 5. *The MSPP algorithm is complete.*

Proof. Let $\mathcal{P}$ denote the set of all ε-feasible, loopless FIP in $\mathcal{T}$ starting at n_{start}. Clearly, the cardinality of $\mathcal{P}$ is finite. When the algorithm backtracks, the node from which it backtracked is marked as visited and hence cannot be selected again until the algorithm backtracks one step farther. This prevents the algorithm from visiting a partial path candidate more than once. As a result, the MSPP algorithm considers every element of $\mathcal{P}$ as a solution candidate at most once, thus it terminates in finite time.

Suppose now that there exists an ε-feasible FIP $\pi \in \mathcal{P}$ from n_{start} to n_{goal}. Suppose, for the sake of contradiction, that π is not found by the MSPP algorithm. It follows that either the MSPP algorithm returns a different ε-feasible FIP $\pi' \in \mathcal{P}$ or it backtracks from n_{start} and reports failure (see Line 15 in Algorithm 1). Suppose that the MSPP algorithm backtracks from n_{start}, say, at iteration i. It then follows that $\pi^i_{\text{start}} = (n_{\text{start}})$. However, every FIP path has n_{start} as its element, and hence $\pi^i_{\text{start}} \subset \pi$. The last expression implies, however, that π^i_{start} can be extended to n_{goal} using a ε-feasible FIP, namely, π. From Proposition 4 it follows that the algorithm does not backtrack from n_{start}, a contradiction. Hence the algorithm returns the ε-feasible FIP π'. □

2.3.2 Complexity

In the following, let $\mathcal{V}$ denote the set of vertices of $\mathcal{G}_i$ and let $\mathcal{E}$ denote the set of edges of $\mathcal{G}_i$.

The reduced graph $\mathcal{G}_i$ keeps nodes of size 1 in a sphere of radius α centered at n_{k_i,p_i}, nodes of size 2 in a sphere of radius 2α centered at n_{k_i,p_i}, etc., and nodes of size 2^ℓ in a sphere of size $2^\ell\alpha$ centered at n_{k_i,p_i}. All these spheres contain approximately the same number of nodes S, since

$$S \approx \frac{\text{volume of the sphere of size } 2^k\alpha}{\text{volume of a node at level } k} = \frac{\frac{\pi^{d/2}}{\Gamma(\frac{d}{2}+1)}(2^k\alpha)^d}{2^{kd}} = \frac{\pi^{d/2}}{\Gamma(\frac{d}{2}+1)}\alpha^d \tag{2.16}$$

is independent of k. Hence, while the search space grows exponentially, as 2^ℓ, the number of nodes of the reduced graph only grows linearly, as ℓS. Since the number of nodes per sphere grows exponentially with the dimension d, it follows that the number of nodes in $\mathcal{G}_i$ is linear in the number of levels of the tree ℓ, and exponential in the number of dimensions d, that is,

$$|\mathcal{V}| = O(\ell 2^d). \tag{2.17}$$

As a reference for comparison, to solve the same problem on a uniform grid, the graph would have $O(2^{\ell d})$ vertices.

Finding the reduced graph nodes

The complexity associated with finding the nodes of the reduced graph depends on the number of nodes of the tree visited. The nodes visited but not selected are parents of the selected nodes since the children are not visited when their parent is selected. It can be easily shown that the total number of visited but not selected nodes is less than the number of selected nodes. Hence, the total number of nodes visited is less than $2|\mathcal{V}|$, so the complexity of this part of the algorithm is $O(|\mathcal{V}|)$.

Finding adjacency

Each pair of vertices is tested for adjacency, so $|\mathcal{V}|(|\mathcal{V}|-1)/2$ tests are executed. The complexity of this step is therefore $O(|\mathcal{V}|^2)$.

Finding the shortest path

The A* algorithm is used to find the shortest path, so the complexity is $O(|\mathcal{E}|+|\mathcal{V}|\log|\mathcal{V}|)$ but $|\mathcal{E}|$ is bounded by a linear function of $|\mathcal{V}|$ because the number of neighbors of a given hypercube is upper bounded by the number of faces of the hypercube times the maximum number of neighbors with respect to one face, $2d \times 2^{(d-1)\ell}$. Hence the complexity of this step is $O(|\mathcal{V}|\log|\mathcal{V}|)$.

The three previous procedures are executed once per iteration. The total complexity per iteration is the complexity of the most complex operation, thus the overall complexity per iteration is $O(|\mathcal{V}|^2)$.

2.3.3 Algorithm Parameter Tuning

Three parameters can be tuned in the MSPP algorithm and determine its performance: the maximum depth of the tree ℓ, the threshold α used to calculate the reduced graph, and the obstacle threshold ε.

Maximum number of levels of the tree ℓ

This parameter determines at which level of detail the world map is used and subsequently the resolution of the smallest nodes of the resulting path. The complexity of each iteration is $O(|\mathcal{V}|^2)$. From (2.17), $|\mathcal{V}|$ is of order $O(\ell 2^d)$. As the search space grows exponentially, the complexity of each iteration grows only quadratically.

Decomposition parameter α

The path constructed by the algorithm should be a finest information path. This is achieved by choosing only the finest information nodes to be part of the path. To ensure that any chosen node represents finest information, it is sufficient that all the neighbors of n_{k_i,p_i} in $\mathcal{G}_i$ are leafs of $\mathcal{T}$. The following proposition gives a condition on the parameter α ensuring that the neighbors of n_{k_i,p_i} on $\mathcal{G}_i$ are leafs of $\mathcal{T}$.

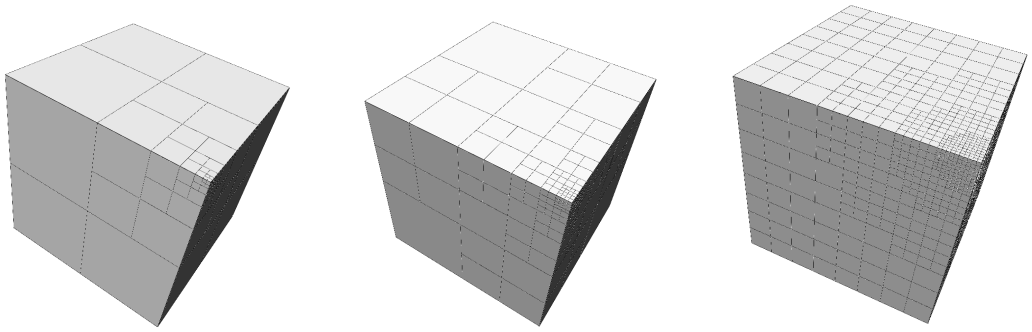
Proposition 6. *Let* $\alpha \geq \sqrt{d}/2$*. Then the selected nodes neighboring the current node are finest information nodes, that is, they are leafs of* $\mathcal{T}$*.*

Proof. Let n_{k_i,p_i} be the current node and let $n_{m,q}$ be one of its neighbors. To guarantee that $n_{m,q}$ represents the finest information we need (2.9) to be false for any value of m, so that $n_{m,q}$ will be selected only if it is a leaf of $\mathcal{T}$ (line 1 of `GetReducedGraphVertices`). The nodes $n_{m,q}$ and n_{k_i,p_i} are neighbors, so $\|q - p_i\|_\infty = 2^{m-1} + 2^{k_i-1}$, and there exist a unique $j \in [1, d]$ such that $(|q - p_i|)_h < 2^{m-1} + 2^{k_i-1}, \quad \forall h \neq j$. Then $\|q - p_i\|_2 < \sqrt{d}\|q - p_i\|_\infty = \frac{\sqrt{d}}{2}(2^m + 2^{k_i})$, and $\|q - p_i\|_2 - \frac{\sqrt{d}}{2}2^{k_i} < \frac{\sqrt{d}}{2}2^m$, from which it follows that by choosing $\alpha \geq \sqrt{d}/2$, condition (2.9) is always false. □

Figure 2.4 shows the result of different values for α when the point of interest is the corner of a cube ($d = 3$). The number of nodes, and hence also the complexity of each iteration, grows when α gets larger. A large value of α provides a better heuristic, and would result in a smaller number of iterations. However, the potential gains of choosing a very large α are canceled by the ensuing slower iterations.

ε-Obstacle threshold

The parameter ε determines the minimum probability by which a node is considered to be an ε-obstacle. This parameter does not change the complexity of the algorithm, but it may lead to rejecting more path candidates, which would imply longer execution time until the first path to the goal is found.



(a) $\alpha = 1$
49 nodes

(b) $\alpha = 2$
217 nodes

(c) $\alpha = 5$
2377 nodes

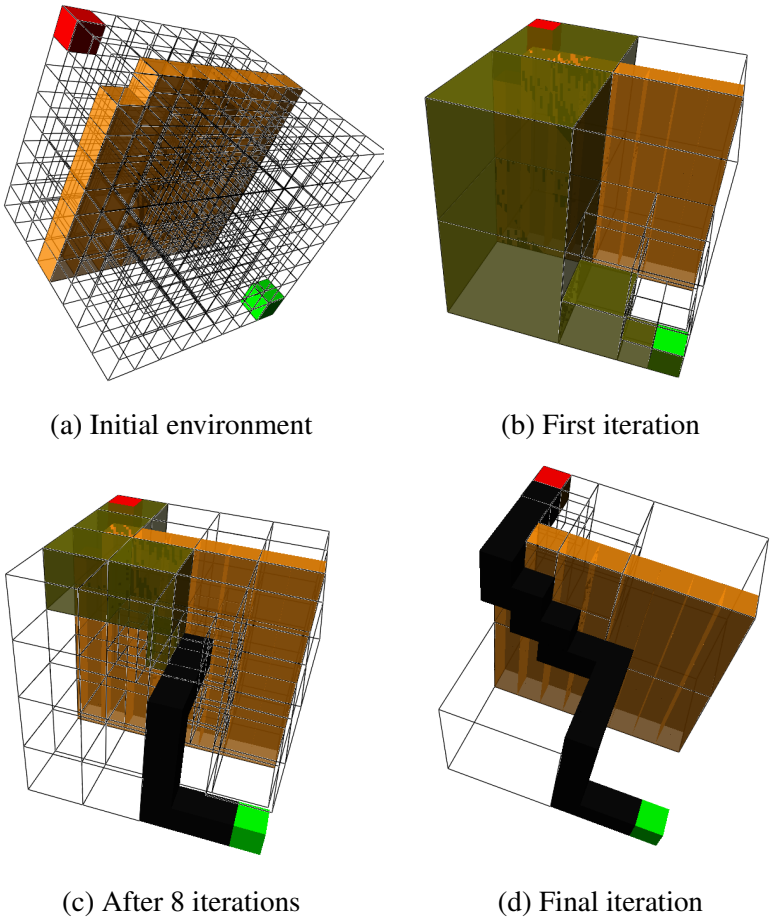
Figure 2.4: Reduced graph for different values of α. The original space contains 287,496 nodes.

2.4 Results

In this section, we present the results of the MSPP algorithm applied to a simple scenario, and we compare the MSPP algorithm with the A* algorithms.

2.4.1 Simple Scenario

In this scenario, a 3D map is given, and the MSPP algorithm is utilized to find an obstacle-free path between two given points in the map. The results of the execution of the algorithm are shown in Figure 2.5. This simple environment shows how the algorithm progresses through the search space to find a path to the goal. The first image shows the initial environment with a complete expansion of the octree structure. The starting node is shown in green and the goal node is shown in red. The obstacles (in yellow/orange) cover an entire wall except for one small hole to go through. The second image shows the results of the first iteration. The reduced graph $\mathcal{G}_i$ is shown in black grid with fine resolution around the starting point and coarser information farther away, along with the shortest path π_i^{goal} found on $\mathcal{G}_i$ (in dark green). After 8 iterations, we also see the partial path candidate π_{start}^i from the start to the current point (in black). At the final iteration, $\mathcal{G}_i$ does not cover the entire space any more, since the nodes containing part of π_{start}^i have been excluded, as described in Section 2.2.1.



(a) Initial environment (b) First iteration

(c) After 8 iterations (d) Final iteration

Figure 2.5: Iteration steps of the algorithm applied to a simple example.

2.4.2 Comparison with the A* Algorithm

The MSPP algorithm is compared to the A* algorithm using three different environments, with different numbers of levels in the tree. Environment 1 has unstructured obstacles with a high density. Environment 2 has structured obstacles with a low density. Environment 3 has unstructured obstacles with low density.

Table 2.1 shows the ratio of the run time of the MSPP algorithm versus the run time of A*. On the small environment, the MSPP algorithm has similar performance than the A* algorithm. As the number of nodes in the environment increases, the MSPP approach shows significant performance enhancement. The run time ratio decreases by an order of magnitude for each level added to the tree.

Table 2.1: Ratio of the running time of the MSPP algorithm versus A*. Environment 1 has unstructured obstacles with a high density. Environment 2 has structured obstacles with a low density. Environment 3 has unstructured obstacles with low density. The number of nodes 512, 4 096 and 32 768 correspond respectively to 3, 4 and 5 depth levels in the tree data structure.

	Environment 1	Environment 2	Environment 3
512 nodes	1.5	1	2
4 096 nodes	0.17	0.25	0.89
32 768 nodes	0.017	0.026	0.091

2.5 Application to a Mobile Robot

2.5.1 Maps Created from Vision Sensor

In this section, we describe how to build a 3D multi-resolution volumetric occupancy map of an environment from camera images, similarly as in [72]. Given a sequence of camera images, we first obtain a camera path and a sparse 3D reconstruction by performing visual SLAM [73, 74]. We model the 3D environment with the data structure described in Section 2.1. However, unlike the traditional occupancy mapping work of [39, 75], we do not assume that each voxel's occupancy $o_{n_{k_j,p_j}}$ is independent of the other voxels. Instead, we form a 3D conditional random field (CRF) [76] over the voxel occupancy states $\{o_{n_{k_j,p_j}}\}$ that enforces spatial regularization over neighboring voxels as follows

$$P(\{o_{n_{k_j,p_j}}\}|\mathcal{D}) = \frac{1}{Z(\mathcal{D})} \prod_j \psi_u(o_{n_{k_j,p_j}}) \prod_{j,m \in \mathcal{N}} \psi_p(o_{n_{k_j,p_j}}, o_{n_{k_m,p_m}}) \tag{2.18}$$

where $Z(D)$ is the partition function over the observed data $\mathcal{D}$, and ψ_u, ψ_p are unary and pairwise potentials [76] described below. The unary potential $\psi_u(o_{n_{k_j,p_j}})$ is defined over each voxel occupancy state $o_{n_{k_j,p_j}}$. We use the sparse point cloud obtained from the visual SLAM pipeline as range measurements to update the unary terms in the same way as laser range measurements are treated in traditional occupancy mapping framework [75, 39]. In (2.18) $\psi_p(o_{n_{k_j,p_j}}, o_{n_{k_m,p_m}})$ is the pairwise potential, enforcing label consistency between two neighboring voxels n_{k_j,p_j} and n_{k_m,p_m} falling into a given neighborhood. We use a Potts

potential [76] for ψ_p, which takes a higher value when labels are the same, and has a lower value when they are different, thus penalizing dissimilar labels across neighboring voxels. The final occupancy map is obtained by Maximum a Priori (MAP) inference [77] over the CRF in (2.18).

Figure 2.6 shows the results of the map creation and the planning at the same time applied to the CamVid [78] and Leuven [79] datasets. The CamVid and Leuven datasets are created from camera images captured from the perspective of a driving automobile with a database of ground truth labels that associate each pixel with a semantic class. On the CamVid dataset, the result of the planning is trivial since the path is a straight line. The Leuven dataset is a bit more challenging and demonstrates some of the potential pitfalls of having a short fine resolution horizon. In particular, the resulting path is not smooth and exhibits several zig-zags. Indeed, if the vehicle is far from the wall, the wall will not be resolved as an obstacle until the robot comes closer. This may create a path that keeps zigzagging near the wall as the robot tries different alternatives. Nonetheless, the robot eventually finds a path to move forward. A remedy to this problem is to use a larger value of α, which would result in detecting the walls earlier but, at the same time, would slow down the algorithm.

(a) CamVid (b) Leuven

Figure 2.6: Results of the planning on the maps reconstructed from the camera images.

2.5.2 Real-Time Application with Unknown Space Exploration

In this section, we present another application of the MSPP algorithm in which the robot is equipped with a laser range sensor and has to reach a known goal while navigating inside an a priori unknown environment. We use a real-time simulation environment which creates strict constraints on runtime. The map is build from the sensor measurements using Octomap [80], and it uses an incremental method that is fast enough for real time implementation. The MSPP algorithm is used to plan on the partially unknown map, and replanning is done when obstacles are detected along the current planned path.

The simulator used is the Gazebo simulation environment associated with the Robotic Operating System (ROS) for communication between the different modules. The simulator provides the ground truth representation of the world and integrates the dynamics of the robot based on the received commands. Noisy laser measurements are generated at 2Hz and the pose of the robot is sent at 100Hz. The Octomap server creates the tree $\mathcal{T}$ using the measurements received, and sends the new map to the planner. The planner checks whether the current planned path is ε-feasible. If it is not ε-feasible it replans until it finds one, and sends the solution to the path tracker as a set of waypoints. Finally, the waypoint tracker generates the motor commands from the current robot pose and the computed waypoints. The waypoint tracker uses proportional feedback control to first align the robot with the waypoint and then move towards that waypoint. A waypoint is considered to have been reached when the distance to the waypoint is below a given threshold, at which time the next waypoint of the path becomes the waypoint to be tracked.

The world used for the simulation is shown in Figure 2.7. The robot starts at the center and the goal is to reach the the red ball.

The path is recalculated when a new goal is assigned, or the current planned path is not ε-feasible any more on the updated map. The computed path is subsequently smoothed and is fed to the trajectory tracking module. Figure 2.8 shows the result of the path smoothing.

Figure 2.8 depicts some key frames of the results. The benefits of using a multi-scale

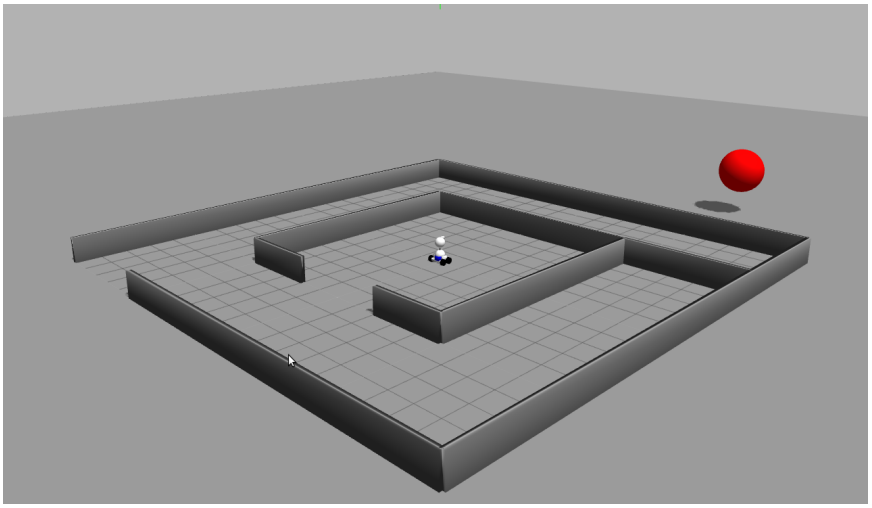
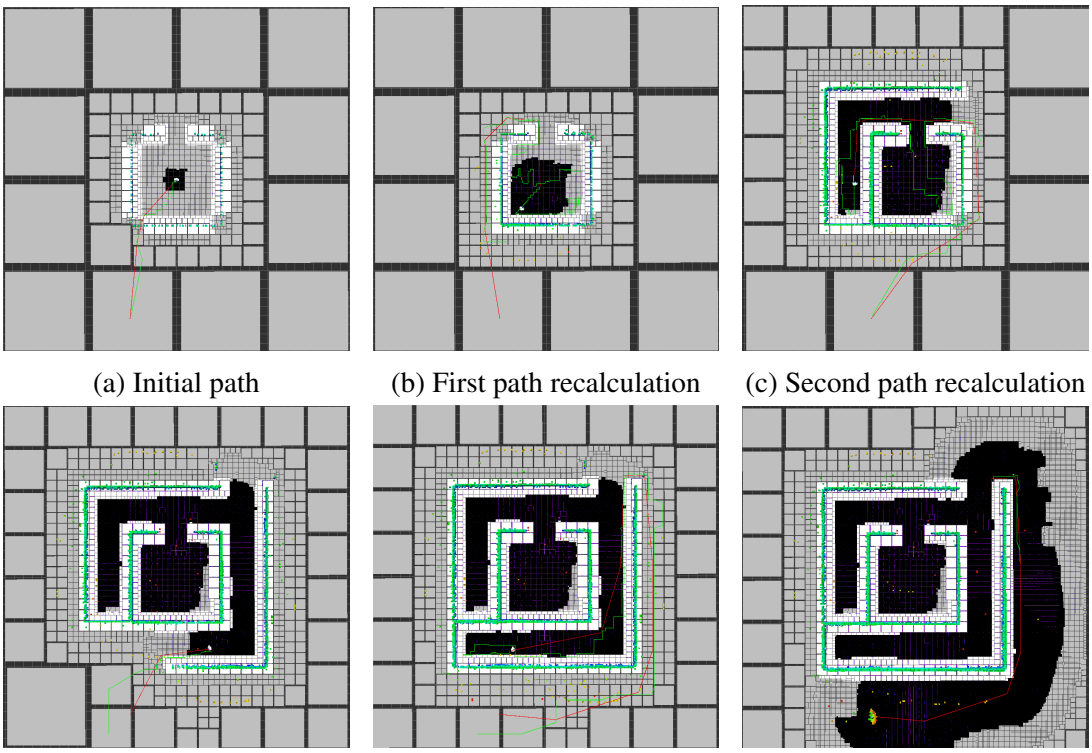


Figure 2.7: Maze used for the simulation, the red ball represents the goal. The robot starts at the center of the maze.

representation can also be seen here since the unknown space is represented by nodes representing large regions of the environment, reducing the size of the data kept in memory. The colored cubes correspond to points measured by the laser sensor, and clearly show the walls. Note that some colored cubes appear on the map but do not correspond to any object of the real world, these elements are due to noisy measurements.

2.6 Fast Neighbors Computation - MSPP-FN

In the MSPP algorithm, neighbors for the reduced graph are computed by testing whether every pair of vertices satisfies the neighborhood properties. As shown before, this step is the bottleneck during each iteration, having complexity $O(|V|^2)$, where $|V|$ is the number of vertices. A new way to compute neighbors by reducing the complexity from $O(|V|^2)$ to $O(|V| \log |V|)$ proceeds as follows; for each vertex, neighbor candidates are generated, and the existence of these candidate vertices in $\mathcal{G}_i$ is verified. The details are given next. MSPP-FN will be used in the rest of the paper to refer to the MSPP algorithm using the fast neighbor computation.



(a) Initial path (b) First path recalculation (c) Second path recalculation

(d) Third path recalculation (e) Fourth path recalculation (f) Goal reached

Figure 2.8: Results of the mapping and planning simulation.
Algorithm raw solution in green, smoothed solution in red. White cells indicate obstacles, black cells are free space and grey is unknown.

2.6.1 Tree Data Structure for Vertices

In order to do fast searches over the vertices of the reduced graph, we keep them in a tree structure. Let $\mathcal{T}_i$ define this tree structure. As the original tree $\mathcal{T}$ is traversed to select nodes for $\mathcal{G}_i$, $\mathcal{T}_i$ is constructed by copying every element of $\mathcal{T}$ traversed by the selection process, except for ε-obstacles. $\mathcal{T}_i$ is then a tree with the same structure as $\mathcal{T}$, but its branches are shorter. In other words, $\mathcal{T}_i$ is a pruned version of $\mathcal{T}$ whose leave nodes are the vertices of $\mathcal{G}_i$.

Note that, for implementation, $\mathcal{T}_i$ does not change significantly between two consecutive iterations, so it is computationally cheaper to modify $\mathcal{T}_{i-1}$ than to create a new data structure at each iteration. Memory allocation is the most expensive operation when creating new nodes. Modifying $\mathcal{T}_{i-1}$ allows to only have to allocate memory for nodes of $\mathcal{T}_i$

that did not exist in $\mathcal{T}_{i-1}$. The copy process is then modified to add nodes only if they do not already exist, and remove excessive nodes when reaching a node corresponding to a vertex of $\mathcal{G}_i$. The pseudo-code is given in Function `GetRGFastNeighbor`. The vertex list is also removed since the information is already contained in $\mathcal{T}_i$. The function `GetRGFastNeighbor` is called with the root of $\mathcal{T}$, the root of $\mathcal{T}_i$ (created during initialization) and the current node n_{k_i,p_i}.

Function GetRGFastNeighbor($n_{k,p}$, $t_{k,p}$, n_{k_i,p_i})

```
1  Function GetRGFastNeighbor (n_{k,p}, t_{k,p}, n_{k_i,p_i})
       Data: Node n_{k,p} (in T), Node t_{k,p} (in T_i), Current node n_{k_i,p_i}
2      if (‖p − p_i‖_2 − (√d/2) 2^{k_i} ≥ α 2^k OR isLeaf(n_{k,p})) AND
        doesNotContainPath(n_{k,p}) then
3          if n_{k,p} is not a ε-obstacle then
4              Remove all descendants of t_{k,p};
5          else
6              Remove t_{k,p} and its descendants;
7      else
8          foreach (m, q) index of children of (k, p) do
9              if t_{m,q} does not exist then
10                 Create t_{m,q} child of t_{k,p};
11             GetRGFastNeighbor (n_{m,q},t_{m,q},n_{k_i,p_i});
```

2.6.2 Same Size Neighbors

Generating neighbors is simple when the nodes have the same size, so we will consider this case first. Given a node $n_{k,p}$, we want to find all its neighbors having the same size that correspond to vertices of $\mathcal{G}_i$. Same size implies the same depth in $\mathcal{T}$, so every neighbor will have the same depth index k. Also, the neighbor conditions and the fact that the nodes are centered on a grid, imply that only one dimension of the position vector can be changed at a time, that is, the neighbors' positions $p_{\text{nhb},i}$ can only be

$$p_{\text{nhb},i} = p + 2^k b_i, \quad 1 \leq i \leq 2d, \tag{2.19}$$

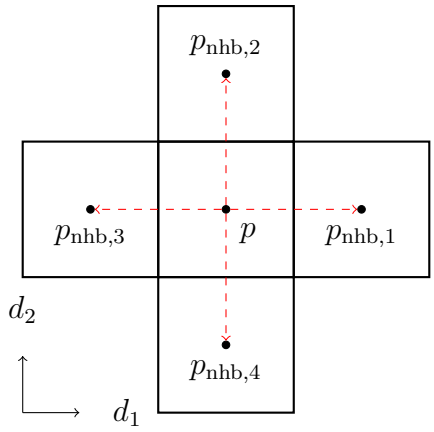


Figure 2.9: Generating neighbors for the construction of $\mathcal{G}_i$. Same size neighbors case: $H(n_{k,p})$ is the center square and the $p_{\text{nhb},i}$ are the generated position candidates for the neighbors.

with

$$b_i = \begin{cases} d_i & \text{if } i \leq d, \\ -d_i & \text{otherwise,} \end{cases}$$

where d_i is the i^{th} vector of the standard basis of $\mathbb{R}^d$. If $p_{\text{nhb},i}$ is within the bounds of the search space, $n_{k,p_{\text{nhb},i}}$ is in $\mathcal{T}_i$ and $n_{k,p_{\text{nhb},i}}$ is a leaf of $\mathcal{T}_i$, then $n_{k,p_{\text{nhb},i}}$ is a valid neighbor of $n_{k,p}$. Figure 2.9 shows in the center the hypercube corresponding to $n_{k,p}$ and the neighbor candidates around it. The red arrows represent the vectors $2^k b_i$.

Searching the tree $\mathcal{T}_i$ can be done on average in $O(\log |V|)$, and the number of candidates to check is $2d$.

2.6.3 Larger Neighbors

Consider now the case of finding the larger neighbors of $n_{k,p}$. The previous result can still be used, but it will generate points inside larger neighbors instead of their positions. The search through the tree works as follows. It starts with the root of the tree, representing the entire environment, as the current node. As long as the current node has children (recall that our data structure assumes that they either all exist or none of them exists), the child whose hybercube contains the searched point $p_{\text{nhb},i}$ is selected as the current node. That is, at each

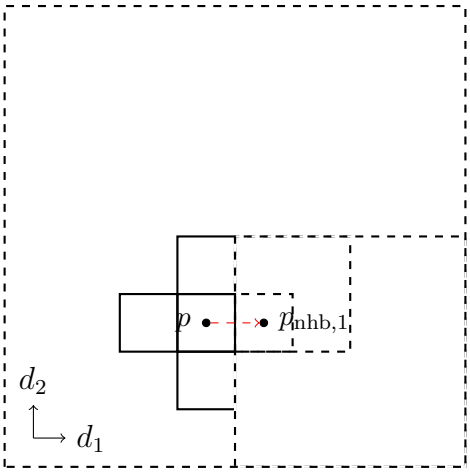


Figure 2.10: Generating neighbors for the construction of $\mathcal{G}_i$. Larger neighbors case: $H(n_{k,p})$ is the smaller square around p and $p_{\text{nhb},1}$ is the first generated position candidate for the neighbors. The dashed squares represents the hypercubes corresponding to nodes visited during the search for $p_{\text{nhb},1}$ in the tree.

step, the search process selects the node at the next level of resolution whose hypercube contains $p_{\text{nhb},i}$. The search stops if either the current node is at $p_{\text{nhb},i}$ or the current node does not have children. At the end of the process, the current node is a neighbor of $n_{k,p}$ and if it is a leaf, then it is also a vertex of $\mathcal{G}_i$. A larger node could contain p and then not be a neighbor, but since $n_{k,p}$ exists, that node would have children and the search process will never stop in such a situation.

Figure 2.10 shows, in dashed lines, the last four nodes that would be explored while searching for $p_{\text{nhb},1}$. If $n_{k,p_{\text{nhb},1}}$ does not exist, the algorithm will stop at one of its ancestors, which will be a neighbor of $n_{k,p}$. Note that the search cannot stop at the largest ancestor shown, since it contains $n_{k,p}$, so all the children exist.

2.6.4 Smaller Neighbors

The last case to consider is when there are smaller neighbors of $n_{k,p}$. The search for $p_{\text{nhb},i}$ in $\mathcal{T}_i$ will return a node that is not a leaf. In this case, the exploration of children of $p_{\text{nhb},i}$ can lead to the neighbors. Note that $p_{\text{nhb},i}$ was generated by moving in the direction b_i, but since the neighbors are smaller, the move was too large, and hence neighbors of $n_{k,p}$ are leaf nodes, descendant of $p_{\text{nhb,i}}$ in the direction $-b_i$.

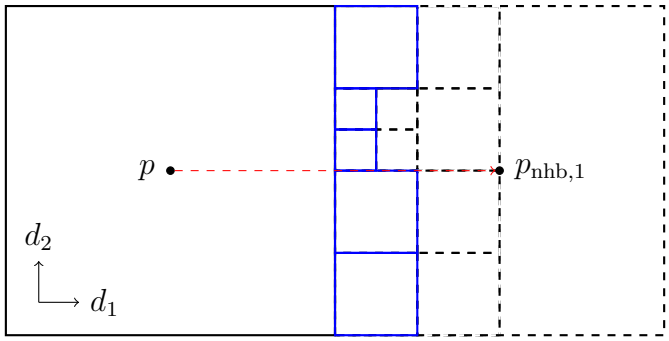


Figure 2.11: Generating neighbors for the construction of $\mathcal{G}_i$. Smaller neighbors case: $H(n_{k,p})$ is the left square and $p_{\text{nhb},1}$ is the first generated position candidate for the neighbors. The larger dashed square is $H(n_{k,p_{\text{nhb},1}})$ and the blue squares correspond to the descendant of $n_{k,p_{\text{nhb},1}}$ that are neighbors of $n_{k,p}$.

Figure 2.11 shows what happens for smaller neighbors. The larger dashed square is the hypercube corresponding to the candidate neighbor $p_{\text{nhb},1}$, but that node is not a leaf, so it is not a vertex of $\mathcal{G}_i$. Exploring its children (until leaf nodes) in the direction $-b_1 = -d_1$, will lead to all its descendants that are neighbor with $n_{k,p}$, and in $\mathcal{G}_i$, since they will be leaf nodes. The neighbors are drawn in blue in Figure 2.11.

2.6.5 Computing all Neighbors in $\mathcal{G}_i$

When looking for all neighbors, nodes can be treated in any order, in particular, from smallest to largest. For the smallest nodes, all neighbors are larger. If all smaller nodes have been treated before, for a given node $n_{k,p}$, the smaller neighbors will already have been found and the only information missing is the larger nodes. All neighbor pairs can then be found by looking for larger neighbors for each node ordered from the smallest to the largest. Finding larger and same size neighbors is done in $O(\log |V|)$ for each of the $|V|$ nodes, so finding all neighboring pairs in $\mathcal{G}_i$ is then be done in $O(|V| \log |V|)$.

2.6.6 Computing all Neighbors of a Given Node $n_{k,p}$

Finding all neighbors of a given node $n_{k,p}$ can be done using the pseudo-code in Function `findNeighbors`. For each direction b_i, we compute the candidate neighbor position $p_{\text{nhb},i}$ and search in $\mathcal{T}_i$ for the corresponding node. If the node is a leaf, it means that a larger

or same size neighbor has been found; otherwise, the leaf descendants, in the direction $-b_i$, of the node found are smaller size neighbors.

Function findNeighbors($n_{k,p}$)

```
1 Function findNeighbors (n_{k,p})
      Data: Node n_{k,p}
2     neighbors=∅;
3     foreach i in [1, 2d] do
4         n_{m,q}=find(T_i,p_{nhb,i});
5         if isLeaf(n_{m,q}) then
6             neighbors=neighbors ∪ n_{m,q};
7         else
8             addLeafInDir(n_{m,q}, -b_i, neighbors);
9     return neighbors;
```

Function addLeafInDir($n_{k,p}$, b, $neighbors$)

```
1 Function addLeafInDir (n_{k,p}, b, neighbors)
      Data: Node n_{k,p}, Direction b, List neighbors
2     foreach i in [1, 2^d] do
3         if b^T e_i > 0 then
4             n=child(n_{k,p},i);
5             if isLeaf(n) then
6                 neighbors=neighbors ∪ n;
7             else
8                 addLeafInDir (n, b, neighbors);
```

2.7 Multi-Scale Path-Planning without Full Information Map - MSPP-S

Although in 2D or 3D geometric workspaces, the multi-scale map is often the result of perception algorithms, this is not always the case. When the search space is the configuration space and it is different from the geometric workspace, computing the multi-scale map might be very expensive, as it requires to analyze every single cell of the map. Furthermore, in some cases, we may only have access to a predicate of whether a point of the search space is an obstacle or not. A robotic arm, for example, is usually parameter-

ized by the position of each joint; given a configuration, the spatial position of each link can be computed, and self-collision or collision with obstacles is checked in the geometric workspace. It is assumed in this section that we have such a predicate, say $isObstacle(s)$, that informs us if a point s of the search space is an obstacle.

In the proposed approach, sampling is used to estimate the obstacle probabilities of the nodes in $\mathcal{G}_i$. Since we are using an estimate instead of the exact node probability values, completeness of the algorithm is not ensured. Note, however, that if a large enough number of samples is drawn, the estimated probabilities will be close to their actual values, and loss of completeness will be very unlikely.

Similarly to the original MSPP algorithm, the proposed algorithm, MSPP-S (as MSPP with sampling), decomposes the space using a grid, which has fine resolution near the current position, and the resolution becomes increasingly coarser farther away. An empty tree data structure $\mathcal{T}_i$ is created to represent that grid. It is empty in the sense that it does not have any information about the obstacles, it is a pure geometric partition of the search space. For each node $n_{k,p}$ of the partition, the predicate can be used for a given number N_{samples} of random points drawn in the search space corresponding to the node. An estimate of the probability of obstacles can then be calculated from those results and used to fill up the tree $\mathcal{T}_i$. The value of the node is approximated by

$$\hat{V}(n_{k,p}) = \frac{\text{Number of obstacles sampled}}{N_{\text{samples}}}. \tag{2.20}$$

Similarly to the data structure used for neighbor checking, it is less costly to modify the data structure from the previous iteration than to recreate a new data structure at each iteration. Moreover, in that case, some information will already exist in the data structure, and sampling only needs to be done for the newly added nodes.

Note that the data structure created is similar to the one created for neighbor checking, hence the same notation $\mathcal{T}_i$. Since only the structure matters for neighbor checking, and not

the actual values, this data structure can also be used for the neighbor checking step.

The pseudo-code for the vertex selection is given in Function `GetRGVerticesWithSampling` and the edges can be computed as described in Section 2.6.5.

Function GetRGVerticesWithSampling

```
1  Function GetRGVerticesWithSampling ()
       Data: Node t_{k,p} (in T_i), Current node n_{k_i,p_i}
2      if (‖p − p_i‖_2 − (√d/2) 2^{k_i} ≥ α 2^k AND doesNotContainPath(t_{k,p}) then
3          Remove all descendants of t_{k,p};
4          if n_{k,p} has not been sampled yet then
5              Sample N_samples in H(n_{k,p});
6              Estimate V̂(n_{k,p});
7      else
8          foreach (m, q) index of children of (k, p) do
9              if t_{m,q} does not exist then
10                 Create t_{m,q} child of t_{k,p};
11             GetRGVerticesWithSampling (t_{m,q}, n_{k_i,p_i});
```

2.8 Minimal Reduced Graph Construction

Constructing $\mathcal{G}_i$ can be costly and only part of the information might be used at each iteration to solve for the shortest path. In this work, it is assumed that the planning problem on $\mathcal{G}_i$ is solved using the A* algorithm although this is not restrictive. In the A* algorithm (see Algorithm 1), nodes are kept in a priority queue, called $OPEN$, ordered by f-values, where $f = g + h$ with g the cost-to-go and h an admissible heuristic to the goal. While $OPEN$ has elements, the first element is removed and for each of the neighbors, if they have not been closed yet, the g-value is updated, and it is added to the $OPEN$ priority queue. The algorithm stops when the first element of the $OPEN$ priority queue is the goal.

In the A* algorithm, knowing the neighbors of a node is only useful when that node is taken out of the $OPEN$ queue. Similarly, the obstacle probability is only needed to calculate the g-value of a node.

By delaying those calculations until the necessary information is required, improvement in execution speed is expected. The following changes allow to save computations in the new algorithm:

- the `ReducedGraph` function only computes the nodes of the reduced graph
- during the A^* algorithm, neighbors of a node are computed when the node is selected from the OPEN priority queue to be explored. If sampling is being used, sampling is only made the first time the g-value is calculated.

At the end, the algorithm will only have calculated the neighbors for the nodes in the $CLOSE$ list, and estimated the obstacle probability for nodes in $OPEN \cup CLOSE$. In the worst case, the A* algorithm will explore every vertex and every edge, so all neighbors will be calculated and all nodes will be sampled similarly to the naïve case. But, in general, the number of neighbors calculated and the number of nodes sampled will be largely reduced compared to the naïve case. Numerical examples in the next section corroborate this hypothesis.

2.9 Probabilistic Bounds of MSPP-S

As the estimate $\hat{V}(n_{k,p})$ is used instead of the actual obstacle probability $V(n_{k,p})$, a bad estimate could lead to missing solutions and hence loss of completeness of the algorithm. In particular, if $\hat{V}(n_{k,p})$ overestimates $V(n_{k,p})$, the node $n_{k,p}$ might wrongly be evaluated as a ε-obstacle which would prevent the algorithm from finding any path passing through $n_{k,p}$, and potentially the only solution, thus breaking the completeness of the algorithm.

In this section, we derive an analytic worst case bound for the probability of failure of the MSPP-S algorithm.

We assume that there is an underlying grid and that each unit cell is either free space or obstacle.

2.9.1 Definitions

To deal with the probability of misevaluating the obstacle probability of a cell, we redefine the notion of obstacles with a threshold γ and the event of wrongly evaluating a node as an obstacle.

Definition 1. *Let $\varepsilon, \gamma > 0$. A node $n_{k,p}$ is a ε, γ-obstacle if*

$$\hat{V}(n_{k,p}) \geq 1 - 2^{-dk}\varepsilon + \gamma. \tag{2.21}$$

Definition 2. *Let $\varepsilon, \gamma > 0$. $M(n_{k,p})$ is the event that the node $n_{k,p}$ is a ε, γ-obstacle and is not a ε-obstacle.*

2.9.2 Bounds on $P\big(M(n_{k,p})\big)$

Proposition 7. *Let $\varepsilon, \gamma > 0$ and n the number of sampled points in a node $n_{k,p}$. Then*

$$P\big(M(n_{k,p})\big) \leq \mathrm{e}^{-2\gamma^2 n}. \tag{2.22}$$

Proof. Suppose $M(n_{k,p})$ is true. Then (2.5) and (2.21) are verified, then from (2.21)-(2.5), we get

$$\hat{V}(n_{k,p}) - V(n_{k,p}) \geq \gamma. \tag{2.23}$$

Suppose that $n_{k,p}$ is composed of N unit cells, including N_o obstacles. Let $\mu = N_o/N$, then $V(n_{k,p}) = N_o/N = \mu$. Uniformly sampling a random point in $n_{k,p}$ is similar to uniformly picking one of the N unit cells, that is, an obstacle is picked with probability $\frac{N_o}{N} = \mu$. Let x_i be the random variable associated with the i^{th} obstacle test. x_i takes value 1 for obstacles and 0 for free space, that is x_i is a Bernouilli trial. Let $n_o = \sum_{i=1}^{n} x_i$. n_o is the number of successes in n Bernoulli trials, so n_o follows a binomial distribution. Using

(2.23), changing variables and using Hoeffding's inequality ((2.27) to (2.28)), we get that

$$
\begin{aligned}
& P\big(M(n_{k,p})\big) && (2.24)\\
\leq\ & P\left(\hat{V}(n_{k,p}) - V(n_{k,p}) \geq \gamma\right) && (2.25)\\
=\ & P\left(\frac{n_o}{n} - \mu \geq \gamma\right) && (2.26)\\
=\ & P\big(n_o \geq n(\gamma + \mu)\big) && (2.27)\\
\leq\ & \mathrm{e}^{-2\gamma^2 n}. && (2.28)
\end{aligned}
$$

□

Proposition 8. *Let a node $n_{k,p}$. If $k \geq k_{\max} = \lceil \frac{1}{d} \log_2 \frac{\varepsilon}{\gamma} \rceil$, then $n_{k,p}$ cannot be a ε, γ-obstacle and $M(n_{k,p})$ never happens.*

Proof. The proof follows easily by the following series of inequalities

$$k \geq k_{\max} = \left\lceil \frac{1}{d} \log_2 \frac{\varepsilon}{\gamma} \right\rceil \geq \frac{1}{d} \log_2 \frac{\varepsilon}{\gamma}. \tag{2.29}$$

It follows that

$$2^{dk} \geq \frac{\varepsilon}{\gamma}. \tag{2.30}$$

Isolating γ, we get

$$\gamma \geq \varepsilon 2^{-dk}. \tag{2.31}$$

Finally, adding one on each side of the inequality, we obtain that

$$1 - 2^{-dk}\varepsilon + \gamma \geq 1. \tag{2.32}$$

□

Proposition 9. *Let a node $n_{k,p}$. If $k \leq k_{\min} = \lfloor \frac{1}{d} \log_2 n \rfloor$, then computing $V(n_{k,p})$ is less expensive than computing $\hat{V}(n_{k,p})$. So ε-obstacles can be used and $M(n_{k,p})$ never happens.*

Proof. The cost of computing $V(n_{k,p})$ is 2^{dk}. The cost of computing $\hat{V}(n_{k,p})$ is n.

$$k \leq k_{\min} = \left\lfloor \frac{1}{d} \log_2 n \right\rfloor \leq \frac{1}{d} \log_2 n. \tag{2.33}$$

It follows that

$$2^{dk} \leq n. \tag{2.34}$$

□

2.9.3 Probability of Failure of MSPP-S

We assume here the MSPP-S algorithm uses ε, γ-obstacles when $k > k_{\min}$ and ε-obstacles otherwise. We also assume that the algorithm evaluates $\hat{V}(n_{k,p})$ at most once for each node, it is evaluated when the value is needed and then the value is kept in memory.

Proposition 10. *Given $\varepsilon, \gamma, n > 0$, an upper bound on the probability of failure of MSPP-S is given by*

$$P(failure) \leq 1 - \left(1 - \exp(-2\gamma^2 n)\right)^{nb_{occ}}, \tag{2.35}$$

where

$$nb_{occ} = \frac{2^{d(\ell - k_{\min})} - 2^{d(\ell - k_{\max}+1)}}{2^d - 1}. \tag{2.36}$$

Proof. $M(n_{k,p})$ can happen for every node such that $k_{\min} < k < k_{\max}$, that is, the maximum number of occurences nb_{occ} of $M(n_{k,p})$ is the number of nodes verifying $k_{\min} < k <$

$k_{\max}$,

$$nb_{occ} = \sum_{k_{\min}<k<k_{\max}} 2^{d(\ell-k)} \tag{2.37}$$

$$= \frac{2^{d(\ell-k_{\min})} - 2^{d(\ell-k_{\max}+1)}}{2^d - 1}. \tag{2.38}$$

If $M(n_{k,p})$ never happens, the algorithm will never discard potential paths, and stay complete, that is,

$$P(success) \geq P\big(M(n_{k,p}) \text{ never happens}\big) \tag{2.39}$$

$$= P\big(\neg M(n_{k,p})\big)^{nb_{occ}}. \tag{2.40}$$

Hence,

$$P(failure) \leq 1 - \big(1 - \exp(-2\gamma^2 n)\big)^{nb_{occ}}. \tag{2.41}$$

□

2.9.4 Upperbound when Multiple Independent Solutions Exist

Suppose there exists Z independant solutions to the planning problem, that is, the search space can de partitionned in Z regions in which there exists at least one solution. Suppose for simplicity that all regions have the same size, $1/Z^{th}$ of the original space.
The probability of failing in one region is then bounded by

$$P(failure \text{ in one region}) \leq 1 - \big(1 - \exp(-2\gamma^2 n)\big)^{nb_{occ}/Z}. \tag{2.42}$$

The probability of the algorithm failing is smaller that the algorithm failing in each region simultaneously since there might exist solutions crossing regions, so

$$P(failure) \leq \Big(1 - \big(1 - \exp(-2\gamma^2 n)\big)^{nb_{occ}/Z}\Big)^Z. \tag{2.43}$$

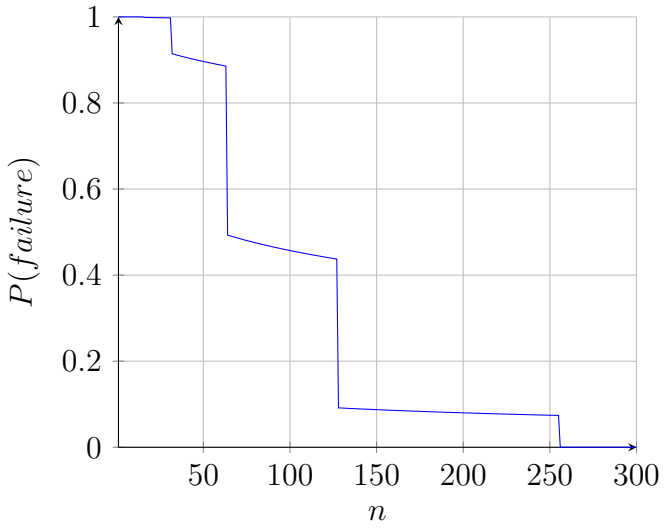


Figure 2.12: Bound on the probability of failure with the parameters $\ell = 5$, $d = 1$, $\varepsilon = 90\%$, $\gamma = 0.35\%$ and $Z = 2$.

Fig 2.12 shows the upperbound on the probability of failure as a function of the number of sampled points n for a given set of parameters of the algorithm. For small n, the probability is very close to 1 since we have a very poor precision in the estimate of the obstacle probabilities. As n grows, the probabilty for every single cell gets better hence the probability of failure decreases. The growth of n also increases the value of $k_{\min}$, thus creating drops in the maximum number of occurences of $M\,(n_{k,p})$ and in the probability of failure. As $k_{\min}$ reaches $k_{\max} - 1$ the maximum number of occurences of $M\,(n_{k,p})$ goes to 0 and the probability of failure of the algorithm then becomes 0.

The upperbounds derived are very conservative in the sense that:

- in most cases, only a subset of the nodes $n_{k,p}$ with $k_{\min} < k < k_{\max}$ are evaluated;
- if $M(n_{k,p})$ happens for a node that is not part of the solution, the solution will still be found by the algorithm;
- if multiple solutions exist but are not independent, the probability is still largely reduced, but not exponentially;
- in typical environments, large areas of free space exist, thus increases the number of possible solutions and largely reduces the probability of failure.

In practice, failure of the MSPP-S algorithm has not been observed.

2.10 Results

2.10.1 Comparison in Random Environments

In this section, we compare the original MSPP algorithm against the proposed extensions and also against the A* algorithm run on a uniform grid. MSPP-FN refers to the variant with the new neighbor test and MSPP-S refer to the variant using sampling (it also uses fast neighbors). Obstacle maps were randomly generated and then used to solve path-planning problems via these three algorithms. The problem was solved for dimensions ranging from 2 to 5 with a tree depth of 5, that is, for search spaces ranging from $2^{2\times5} = 1024$ to $2^{5\times5} \simeq 3 \times 10^7$.

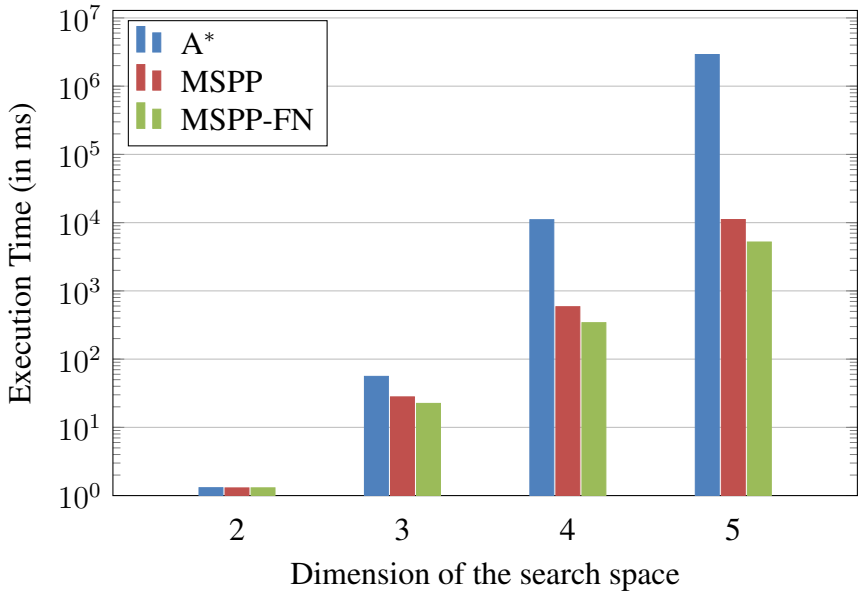


Figure 2.13: Comparison of the execution time of the A*, MSPP and MSPP-FN algorithms. The results are shown in logarithmic scale.

Figure 2.13 shows the average execution time (in log scale) of the MSPP, the MSPP-FN and the A* algorithms on the randomly generated maps. In this figure, the time to create the map is not taken into account in order to compare the pure performance of the planning algorithms, that is, it is just the time to find a path on an already existing map.

For the smaller search spaces, we see very few differences between all the algorithms,

as expected. As the dimension and the size of the search space grow however, the MSPP algorithm becomes much faster than the A*, by more than two orders of magnitude in dimension 5. By the same token, the MSPP-FN algorithm is even faster (by 50%) over the baseline MSPP algorithm in dimension 5.

In Figure 2.14, the cost of creating the map is taken into account. This is done in order to compare the results of using the MSPP-S algorithm. Three algorithms are compared

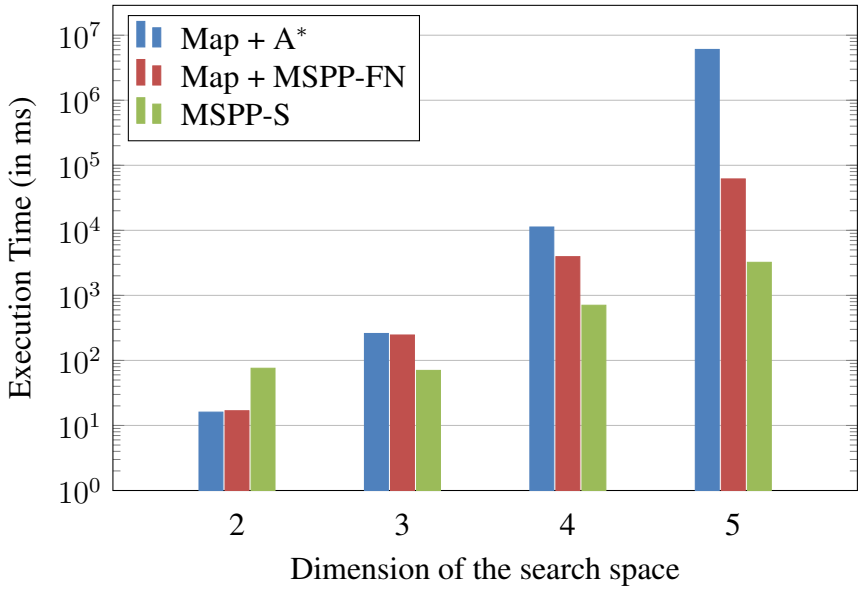


Figure 2.14: Comparison of the time to construct the map and run the A* or MSPP-FN algorithm versus the time to run the MSPP-S algorithm for which a map does not need to be computed. The results are shown in logarithmic scale.

here, the A* algorithm with the construction of the graph, the MSPP algorithm with the construction of the multi-scale map and the MSPP-S algorithm. Similarly to the previous case, on a small search space, there is little or no improvement. As the problem dimension increases, however, the improvement gets much better. The MSPP-S algorithm is three orders of magnitude faster than creating a map and using the A* algorithm, and more than ten times faster than constructing a multi-scale map and using the original MSPP algorithm.

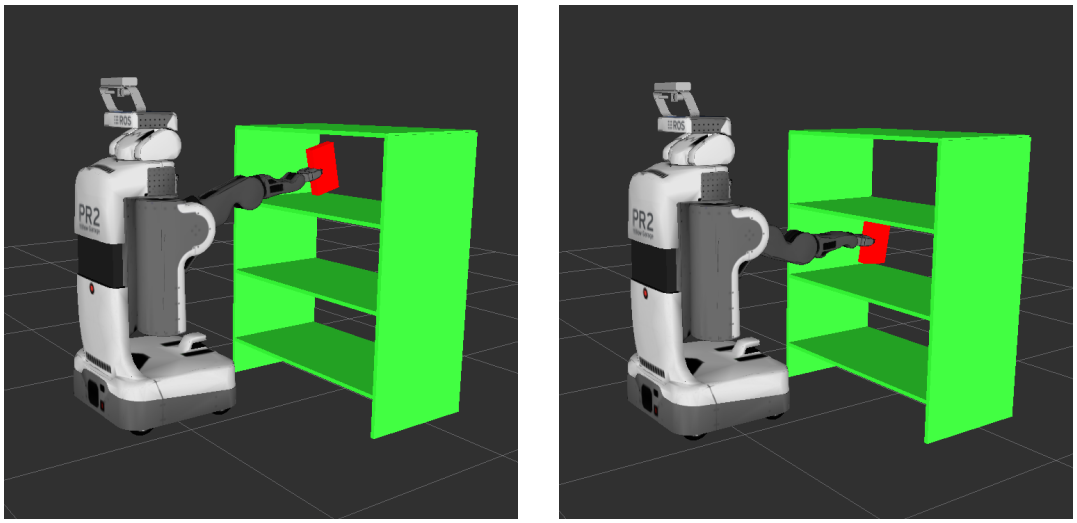


Figure 2.15: Initial and final pose of the planning problem for the PR2 arm.

2.10.2 Application to a Robot Arm

The planning algorithm was used to plan a trajectory for an arm of the PR2 robot. Planning was done in the configuration space using four joints of the arm. Figure 2.15 shows the initial configuration and the desired final configuration; the robot needs to move a book from the top shelf to the second shelf. The depth of the tree was set to 5, creating a search space of size $2^{4\times5} \simeq 3 \times 10^7$.

The path-planning problem was solved three times to compare the variants of the algorithm. The multi-scale map was built by exploring the entire search-space and the MSPP algorithm was used to find the solution. All three algorithms were executed on the same desktop computer running Ubuntu Linux. Building the map was the most time-consuming process. It takes on average 4 minutes and 52 seconds to build the map while solving the path-planning problem takes on average 47 seconds using the MSPP algorithm. Using the MSPP-FN algorithm on the same map, the problem was solved in 4 seconds on average.

2.11 Summary

In this chapter, we presented an n-D multi-scale path-planning algorithm, and a perception algorithm using the same data structure. The algorithm is proven to be complete. The com-

plexity of the algorithm is shown to grow only linearly, while the size of the map grows exponentially. This allows the algorithm to be implemented on large maps without excessive execution times. The proposed algorithm seamlessly integrates multi-scale perception with multi-scale path-planning. Several modifications and extensions to the MSPP algorithm are also presented to increase the computational efficiency of the algorithm. The resulting multi-scale path-planning algorithms, called MSPP-FN and MSPP-S offer several non-trivial improvements over the previous MSPP algorithm. First, the complexity of each iteration of the algorithm is reduced by changing the manner by which the adjacency relationships in the reduced graph are computed at each iteration. Second, the range of applications of the algorithm has been widened, by allowing the use of an obstacle predicate rather than accurate prior knowledge of a full information multi-scale map. This extension results in much fewer requirements in terms of memory allocation and theoretical bounds on the probability of failure were derived. Third, reordering the operations performed by the algorithm allows one to minimize computations by avoiding the computation of information that is not needed during execution. The MSPP algorithm and its variants, MSPP-FN and MSPP-S, were compared and the variants showed runtime improvements by over 50%. Both variants outperform A* by more than two orders of magnitude.

The algorithms were applied to multiple problems to demonstrate the possibility of direct integration with perception, as well as real-time planning on partially unknown maps. The robotic arm example shows how the algorithm can also be used in the configuration space and for larger dimensions.

CHAPTER 3

CONTINUOUS SEARCH SPACES

3.1 The Hypercube Diagonal Experiment (HDE)

In this section, we present a simple numerical experiment that will be used to compare the convergence results of sampling-based algorithms. Let the search space $\mathcal{S}$ be a hypercube of dimension d with each dimension taking values from -1 to 1, namely $\mathcal{S} = [-1, 1]^d$. Assume momentarily that $\mathcal{S}$ is obstacle free. Let the starting point be

$$x_{\text{start}} = [-1, -1, ..., -1], \tag{3.1}$$

and the goal point be

$$x_{\text{goal}} = [1, 1, ..., 1], \tag{3.2}$$

that is, x_{start} and x_{goal} are the two opposite corners of the hypercube $\mathcal{S}$. The cost function c is the length of the path, normalized by $2\sqrt{d}$,

$$c(x_1, x_2) = \frac{\|x_1 - x_2\|}{2\sqrt{d}}, \qquad \forall x_1, x_2 \in \mathcal{S}. \tag{3.3}$$

Clearly, the optimal solution of the HDE is the straight line connecting x_{start} and x_{goal}, that is, the diagonal of the hypercube. The length of the diagonal is $2\sqrt{d}$, so the cost of the optimal solution is $c^* = 1$. Using a normalized cost, such that the optimal solution is independent of the dimension, allows us to easily compare the convergence results across multiple dimensions.

In order to study convergence, we enforce a maximal distance between consecutive elements of the solution path π. In the RRT family, this parameter is often called the range of the algorithm and it is used as the maximum length allowed when creating a new edge

in the tree $\mathcal{T}$. Let the range for the HDE be

$$range = 0.1\sqrt{d}\,. \tag{3.4}$$

Similarly to the cost, the range is normalized, such that the optimal solution can be built with the same number of nodes, independently of the dimension d. Specifically, an optimal solution can be built using exactly 19 nodes spread uniformly between x_{start} and x_{goal} for any dimension d, as can be seen in Figure 3.1.

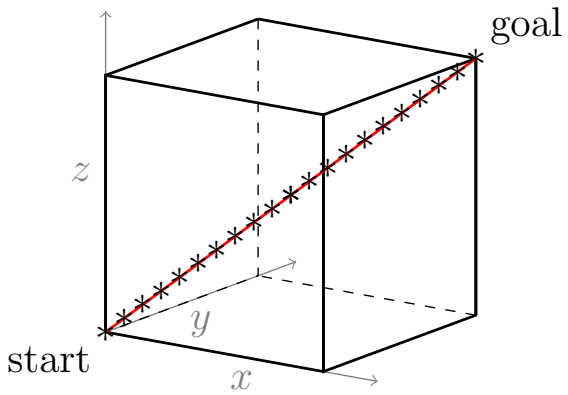


Figure 3.1: The Hypercube Diagonal Experiment in three dimensions. The red line shows the optimal solution and the stars ($*$) show the nodes for the optimal solution with the minimal number of nodes.

3.2 Pure Sampling Strategy

In this section, we analyze the probability of sampling "good" points using a uniform sampling strategy. A "good" sample is a point of the search space that is likely to help the convergence of the algorithm, that is, the point has to be sampled close to the optimal solution. Note that for real applications, the optimal solution is unknown, so it cannot be used to generate samples.

Let π^* be the optimal path from x_{start} to x_{goal} and let $\varepsilon > 0$ be a distance threshold. Define a tube for the path π^* and radius ε to be the set of points within a distance ε from

π^*,

$$tube(\pi^*, \varepsilon) = \left\{ x \in \mathcal{S} \middle| \begin{array}{l} \exists i \in [1, N], \\ distance(x, [\pi^*_{i-1}, \pi^*_i]) \leq \varepsilon \end{array} \right\}, \tag{3.5}$$

where $distance(x, [a, b])$ is the shortest distance between the point x and the segment $[a, b]$. Let also x_{rand} be a random variable, uniformly distributed over the sampling space $\mathcal{X}_s$. Define the event GS ("good" sample) to be the event that x_{rand} is within ε from the optimal path, that is,

$$GS = \{x_{\text{rand}} \in tube(\pi^*, \varepsilon)\}. \tag{3.6}$$

The probability of sampling a "good" point is then equal to the ratio of the volume of $tube(\pi^*, \varepsilon)$ and the volume of $\mathcal{X}_s$,

$$P(GS) = \frac{|tube(\pi^*, \varepsilon)|}{|\mathcal{X}_s|}. \tag{3.7}$$

Consider now the case of the HDE with a uniform sampler within the hypercube. Since the optimal solution π^* is the diagonal of the hypercube and the sampling space $\mathcal{X}_s$ is the entire hypercube, it follows that

$$\begin{aligned} P(GS) &= \frac{2\sqrt{d}V_{d-1}(\varepsilon)}{2^d} && (3.8) \\ &= \frac{2\sqrt{d}\varepsilon^{d-1}V_{d-1}(1)}{2^d} && (3.9) \\ &= \left(\frac{\varepsilon}{2}\right)^{d-1} \sqrt{d}V_{d-1}(1), && (3.10) \end{aligned}$$

where $V_d(r)$ is the volume of a d-ball of radius r. It can be seen that $\sqrt{d}V_{d-1}(1)$ reaches the maximum of $\sqrt{7}\pi^3/6$ for $d = 6$, so

$$P(GS) \leq \frac{\sqrt{7}\pi^3}{6} \left(\frac{\varepsilon}{2}\right)^{d-1}. \tag{3.11}$$

The dominant term in (3.11) is ε^{d-1}. Hence, as ε gets smaller, the probability of sam-

pling "good" points decreases, especially when d is large.

If a heuristic h is available, smarter sampling can be performed. In particular, sampling can be done only in the relevant region, $\mathcal{X}_s = \mathcal{X}_{\text{rel}}(J)$. Assuming J^* is the cost of the optimal solution, we have $J^* \leq J$ and thus $\mathcal{X}_{\text{rel}}(J^*) \subset \mathcal{X}_{\text{rel}}(J)$ and $|\mathcal{X}_{\text{rel}}(J^*)| \leq |\mathcal{X}_{\text{rel}}(J)|$.

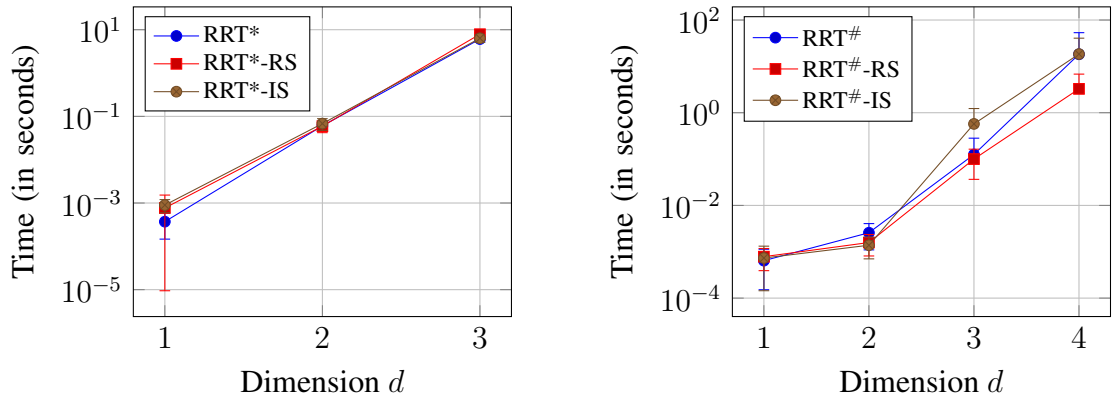
Assume $|\mathcal{X}_{\text{rel}}(J^*)| > 0$. If $|\mathcal{X}_{\text{rel}}(J^*)|$ was equal to 0, the optimal value function would already be known along the optimal path and the problem would thus already be solved. Then

$$
\begin{aligned}
P(GS) &= \frac{2\sqrt{d}V_{d-1}(\varepsilon)}{|\mathcal{X}_{\text{rel}}(J)|} && (3.12)\\
&\leq \frac{2\sqrt{d}V_{d-1}(\varepsilon)}{|\mathcal{X}_{\text{rel}}(J^*)|} && (3.13)\\
&\leq \frac{\sqrt{7}\pi^3}{3|\mathcal{X}_{\text{rel}}(J^*)|}\varepsilon^{d-1}. && (3.14)
\end{aligned}
$$

Thus, if a heuristic is known, smart sampling can be used to improve the probability of "good" samples, but the dominant term is still ε^{d-1}.

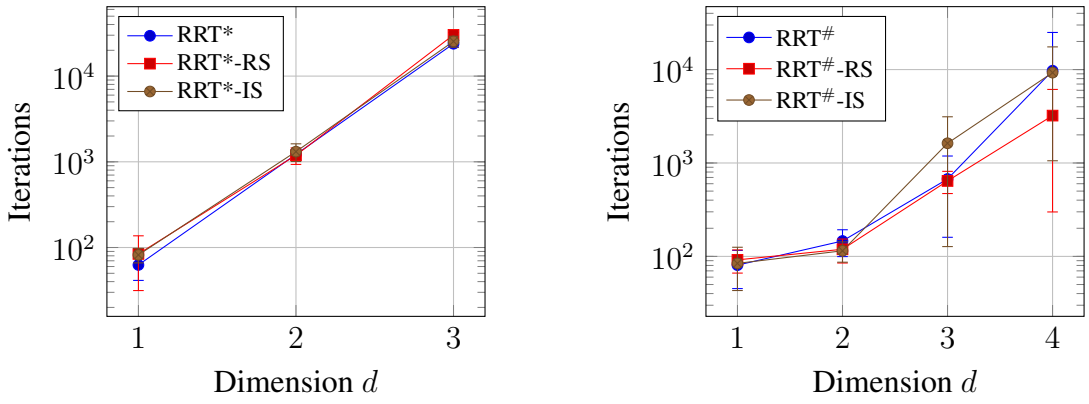
For instance, with $\varepsilon = 0.2$ in the case of two dimensions, the probability upperbound is around 20%, so "good" samples are likely, but in dimension 10, the probability upperbound is $10^{-6}\%$ so "good" samples become a very rare event.

Figure 3.2 shows the number of iterations and the time required for the RRT* and the RRT$^{\#}$ algorithms to converge within 2.5% of the optimal solution for the HDE. The suffixes RS and IS correspond to rejection sampling and informed sampling using a perfect heuristic, that is, a heuristic $h(x_1, x_2)$ providing the actual optimal cost, not just a lower bound for the cost from x_1 to x_2. We can see from these results that smart sampling slightly helps convergence, but even with a perfect heuristic the results are within the same order of magnitude. Moreover, we see an exponential increase in the number of iterations required for convergence, which corresponds to the exponential decrease in the probability of sampling good points seen in (3.14).



(a) RRT* - Time

(b) RRT$^{\#}$ - Time



(c) RRT* - Iterations

(d) RRT$^{\#}$ - Iterations

Figure 3.2: Time and number of iterations to converge within 2.5% of the optimal cost for the hypercube diagonal experiment as a function of the dimension of the problem.

3.3 Optimizing Samples Location

In this section, we analyze how samples can be repositioned in order to minimize the cost without adding more samples, and thus increase the convergence rate. The idea is to minimize the error of the estimated value function with a fixed number of samples. In our case, the value function at $x \in \mathcal{S}$ is the optimal cost to go from x_{start} to x.

Suppose a certain number of samples has been drawn and the information gathered has been stored in a tree $\mathcal{T}$. The tree is a set of nodes with root node, start, corresponding to x_{start} in the search space. Each node $n \in \mathcal{T}$ contains the following information:

- x_n: the corresponding state in the search space $\mathcal{S}$,
- p_n: the parent node of n in the tree,
- $\mathcal{C}_n$: the list of node children of n,
- c_n: the cost from start to node n following the strategy encoded by $\mathcal{T}$,
- $\mathcal{N}_n$: the list of nodes that are the nearest neighbors of n in the search space.

Let $V(x)$, for $x \in \mathcal{S}$ be the value function we want to estimate. That is, $V(x)$ is the minimum cost to reach x starting from x_{start}. Let $\mathcal{S}^*$ be the subset of $\mathcal{S}$ reachable from x_{start}, that is,

$$V(x) \begin{cases} < \infty, & \forall x \in \mathcal{S}^*, \\ = \infty, & \forall x \in \mathcal{S} \setminus \mathcal{S}^*. \end{cases} \tag{3.15}$$

Let also $\hat{V} : \quad \mathcal{S}^* \to \mathbb{R}^+$ be the estimate of the value function built using the tree, defined as

$$\hat{V}(x) = \begin{cases} c_n, & \exists n \in \mathcal{T} \text{ s.t. } x_n = x \\ \min\limits_{\substack{n \in \mathcal{T}, \\ s.t. \\ isFeasible(n,x)}} c_n + c(n,x), & \exists n \in \mathcal{T} \text{ s.t. } isFeasible(n,x) \\ \infty, & \text{otherwise,} \end{cases} \tag{3.16}$$

where $isFeasible(n,x)$ is true if the path from n to x is obstacle-free. In other words, $\hat{V}$ is defined at the location of the samples of the tree by their cost value, and extended to the entire search space using a feasible path connecting to the tree with lowest cost. By construction, all trajectories in $\mathcal{T}$ are feasible, thus

$$\hat{V}(x) \geq V(x), \tag{3.17}$$

because V is the minimum cost to reach x.

The problem is to find the best location for the samples, organized in $\mathcal{T}$, such that $\hat{V}$

estimates V as closely as possible on $\mathcal{S}^*$. The problem can be formulated as

$$\arg\min_{x_n, n\in\mathcal{T}} \tfrac{1}{|\mathcal{S}^*|} \textstyle\int_{\mathcal{S}^*} |\hat{V}(x) - V(x)| \mathrm{d}x \tag{3.18}$$

$$\Leftrightarrow \quad \arg\min_{x_n, n\in\mathcal{T}} \tfrac{1}{|\mathcal{S}^*|} \textstyle\int_{\mathcal{S}^*} (\hat{V}(x) - V(x)) \mathrm{d}x \tag{3.19}$$

$$\Leftrightarrow \quad \arg\min_{x_n, n\in\mathcal{T}} \tfrac{1}{|\mathcal{S}^*|} \textstyle\int_{\mathcal{S}^*} \hat{V}(x) \mathrm{d}x. \tag{3.20}$$

Equation (3.17) is used to go from (3.18) to (3.19), and noticing that the integral of V over $\mathcal{S}^*$ is a constant independent of the samples allows us to simplify the problem to (3.20).

Integrating $\hat{V}$ over $\mathcal{S}^*$ is computationally very expensive, but the integral can be estimated using the samples of the tree, where $\hat{V}$ can be evaluated from,

$$\frac{1}{|\mathcal{S}^*|} \int_{\mathcal{S}^*} \hat{V}(x) \mathrm{d}x \simeq \frac{1}{|\mathcal{T}|} \sum_{n\in\mathcal{T}} \hat{V}(x_n) = \frac{1}{|\mathcal{T}|} \sum_{n\in\mathcal{T}} c_n. \tag{3.21}$$

Since $|\mathcal{T}|$ is constant, the optimization problem is reduced to

$$\min_{x_n, n\in\mathcal{T}} \sum_{n\in\mathcal{T}} c_n. \tag{3.22}$$

By the construction of $\mathcal{T}$, we have

$$c_n = \begin{cases} 0, & \text{if } n = \text{start}, \\ c(x_{p_n}, x_n) + c_{p_n}, & \text{if } n \neq \text{start}. \end{cases} \tag{3.23}$$

We can then compute

$$\sum_{n\in\mathcal{T}} c_n = \sum_{i\in\mathcal{T}} \sum_{j\in\mathcal{T}} nb\_path(i,j) c(x_i, x_j). \tag{3.24}$$

where $nb\_path(i,j)$ is the total number of paths in the tree using the (i,j) edge. Because of the tree structure, there is exactly one path connecting the root of the tree to each node.

Moreover, this is true for any subtree, so

$$nb\_path(i,j) = \begin{cases} 0, & \text{if } p_j \neq i \\ 1 + nb\_des(j) = d_j, & \text{otherwise,} \end{cases} \tag{3.25}$$

where $nb\_des(j)$ is the number of descendants of j in $\mathcal{T}$. The paths going through the edge from i to j are the path to j plus the paths to all the descendants of j. We thus get,

$$\sum_{n \in \mathcal{T}} c_n = \sum_{i \in \mathcal{T}} \sum_{j \in \mathcal{C}_i} d_j c(x_i, x_j). \tag{3.26}$$

By taking the gradient of the previous expression, we find the direction that minimizes the error on the value function as follows

$$\frac{\partial \sum_{n \in \mathcal{T}} c_n}{\partial x_i} = \sum_{j \in \mathcal{C}_i} d_j \frac{\partial c}{\partial x_i}(x_i, x_j) + d_i \frac{\partial c}{\partial x_i}(x_{p_i}, x_i). \tag{3.27}$$

This gradient with respect to the position of the node i in the search space is easily computable, as it depends only on the parent node p_i and the children nodes $\mathcal{C}_i$.

A gradient descent algorithm can then be used in order to optimize the position of the samples as long as the tree $\mathcal{T}$ stays valid, that is, every edge of $\mathcal{T}$ stays feasible and obstacle-free.

This operation can be seen as an a posteriori informed sampling in the sense that the sample positions are updated after having been added to the tree.

3.4 Optimizing Samples Location - Results

Recall the minimization problem, $\min_{n_x, n \in \mathcal{T}} \sum_{n \in \mathcal{T}} c_n$ and note that a trivial solution of this problem is to move every single node to x_{start}, reducing the cost of each node to 0. This solution is of no interest, so the following constraints need to be added to the problem:

- The start and goal nodes cannot be repositioned as they are part of the definition of

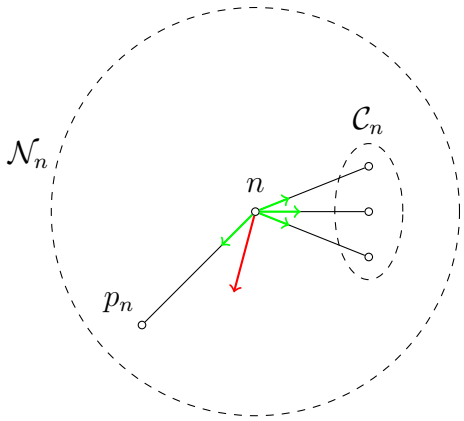


Figure 3.3: Computation of the gradient using local information - The gradients corresponding to the children and the parent node are in green, and the total gradient is in red.

the problem.

- The coverage of the search space should remain unchanged because we do not want to lose the information that has been acquired about the search space through sampling. The leaf nodes of the tree will then be fixed in the search space.

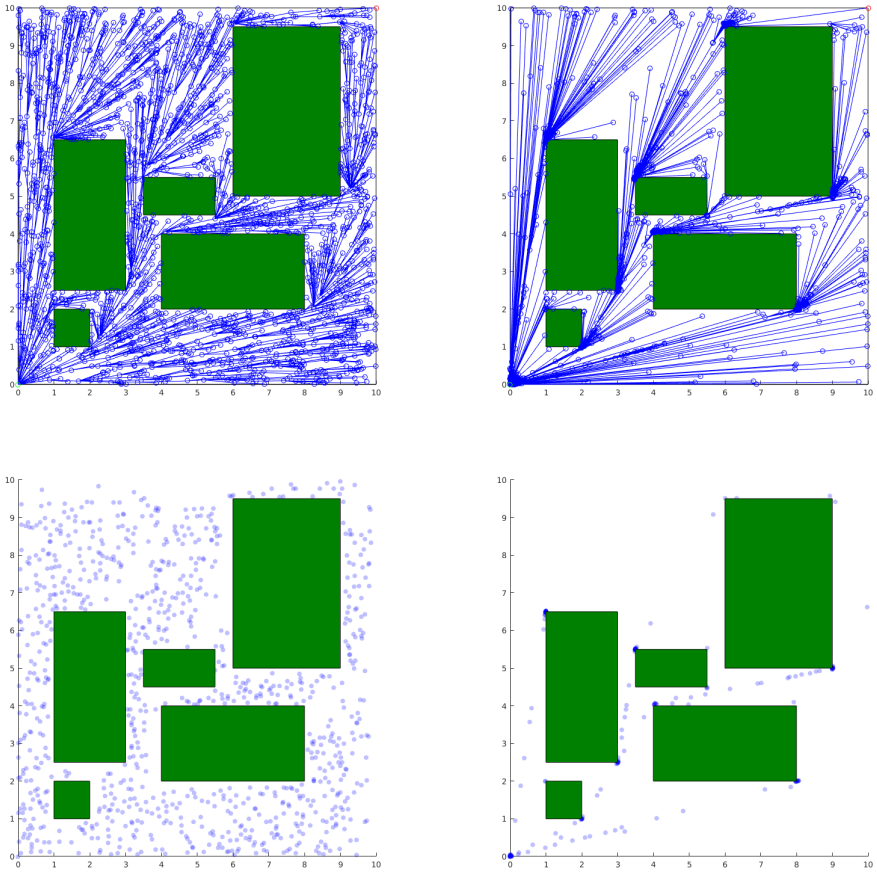
Define the interior nodes of the tree as the set of nodes that are not leaf nodes and are different than start and goal. The gradient descent algorithm is then applied only to the interior nodes of the tree.

Algorithm 3: Gradient Descent

```
1 while gradientDescentTerminationCheck() do
2     foreach n ∈ T do
3         if n == start OR n == goal OR isLeaf(n) then
4             continue;
5         x_temp ← x_n − α ∂(Σ_{n∈T} c_n)/∂n_x;
6         if isFeasible(x_{p_n}, x_temp) AND isFeasible(x_temp, C_n) then
7             x_n ← x_temp;
```

Algorithm 3 shows the structure of the gradient descent. Nothing happens to start, goal and the leaf nodes of $\mathcal{T}$. For the other nodes n, a temporary location x_{temp} is computed from x_n, following the gradient with a scaling factor α. If the new location verifies feasibility of the edges of $\mathcal{T}$, that is, connection between n and its parent p_n and connections

between n and its children $\mathcal{C}_n$, the location of n is updated. The process is iterated until a termination condition is reached, for example a finite number of iterations has been reached or a convergence criterion is satisfied.



Iteration 1 Iteration 100

Figure 3.4: Evolution of the samples during the gradient descent.

Figure 3.4 shows the evolution of the tree during gradient descent. The first row shows the tree structure and the second row shows the position of the interior nodes of the tree. The interior nodes start with an almost uniform distribution over the search space, and as gradient descent is applied, they start to concentrate around the optimal trajectories in each homotopy class. As the gradient descent continues, at iteration 100, the samples get sparser even on the optimal trajectories and start accumulating at specific points of the

environment, namely the corners of obstacles, as expected for this environment.

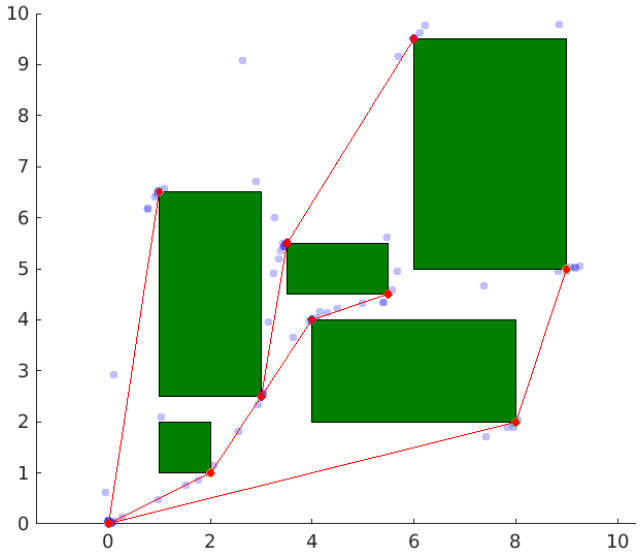


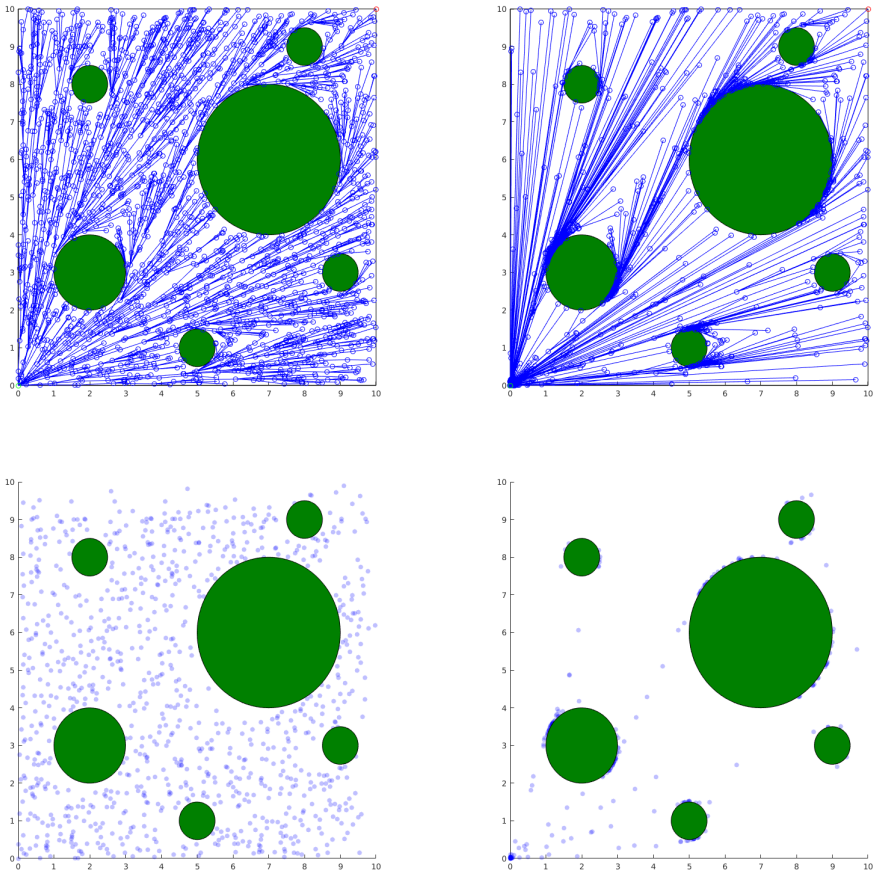
Figure 3.5: Interior nodes position after applying the gradient descent.

Figure 3.5 shows in greater detail the distribution of the interior nodes after applying gradient descent. Circled in red are the accumulation points, and red lines show the tree built with these accumulation points. This tree corresponds to the optimal tree starting from the bottom left corner built on the visibility graph of this environment. That is, the optimal solution from the bottom left corner to anywhere in the search space can be found using this tree and an additional node positioned at the desired goal.

Without any assumption on the type of obstacles, the gradient descent recovered the important points of the environment. In particular, for a shortest Euclidean path with polygonal obstacles, we recovered, as expected, that these points are located at the corners of obstacles, thus finding the optimal tree on the visibility graph given the starting position x_{start}.

Figure 3.6 shows similar results for an environment with circular obstacles. In this case, the samples accumulate on the boundary of the obstacles, which are the points of interest when searching for the shortest path in that type of environment.

Figure 3.7 shows the evolution of the cost optimized by the gradient descent and of the



Iteration 1 Iteration 100

Figure 3.6: Evolution of the samples during the gradient descent with circular obstacles.

best cost to the goal as a function of the number of iterations. Each iteration is composed of two steps:

- Optimize the wiring of the tree, given the samples (in a similar fashion than the LPA* algorithm [27]),
- Optimize the location of the samples, given the tree structure (gradient descent).

The cost after each step is shown on the plots. We can see an overall reduction of the cost for the entire environment, as well as a reduction of the cost to the goal. The goal cost does not exhibit a smooth decrease because it is not the value being optimized, and moving a node that is part of the best path can temporarily increase the cost.

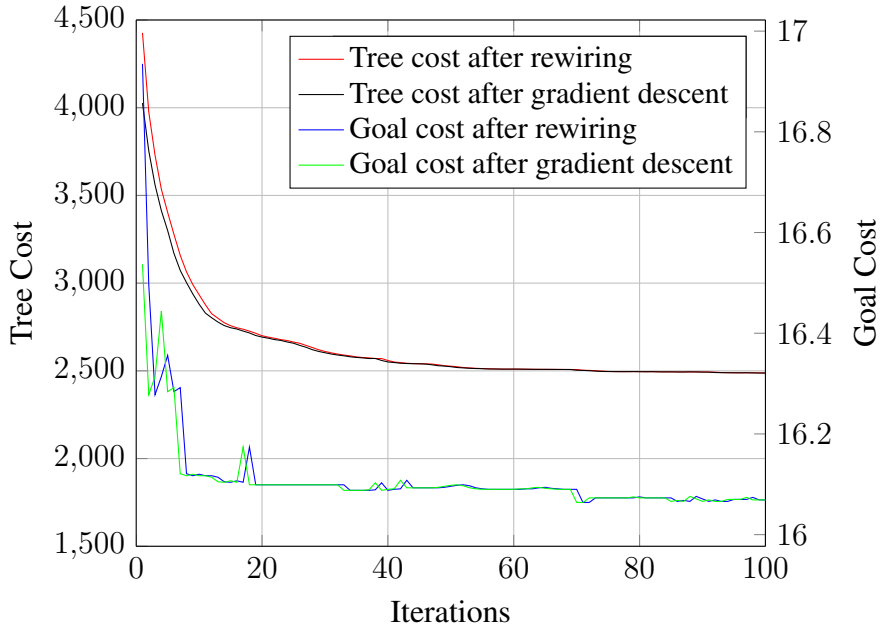


Figure 3.7: Convergence of Optimized Tree Cost and Goal Cost.

3.5 The DRRT Algorithm

In this section we present the Deformable Rapidly-Exploring Random Trees algorithm that merges the idea of optimizing the position of the samples in the framework of RRT-like algorithms.

When samples are added iteratively, there is no need to do a global gradient descent as changes occur only locally. Suppose the tree is already optimized with respect to both wiring of the tree and position of the samples. Then adding a new sample only creates local changes near that sample, and perhaps also down the branch of the tree, if it can be used to improve the cost of other nodes. The branch from the root of the tree to the new sample can be optimized, and thus improve the value function at the new location, before using it to propagate information through the tree.

Each iteration of the DRRT algorithm can be described in three main steps:

- Sample a new point x_{new} and connect it to the tree as a leaf.
- Optimize the location of the nodes on the branch from the root to x_{new}.

- Propagate changes through the tree.

Algorithm 4: The DRRT Algorithm

```
1  T = {start};
2  while drrtTerminationCheck() do
3      Q ← ∅;
4      x_rand ← NewSample();
5      x_near ←Near(x_rand);
6      x_new ←Steer(x_rand, x_near);
7      if isFeasible(x_near, x_new) then
8          n_new ←NewNode(x_new, x_near);
9          OptimizeParent(n_new);
10         b ←Branch(n_new);
11         GradientDescent(b);
12         Q.insert(b);
13         PropagateChanges();
```

The core of the algorithm, shown in Algorithm 4, is similar to the one of RRT$^{\#}$ with an extra step of performing gradient descent. At each iteration, the queue Q, used to order the nodes to be updated, is cleared. A new sample x_new is drawn, and the nearest element x_near in $\mathcal{T}$ is found. The algorithm steers x_new to be within the defined maximum range of x_near. If the path from x_near to x_new is feasible, a new node n_new is created and added to the tree $\mathcal{T}$. The node is initialized with x_near has its parent and its neighbors are computed. Within the neighbors, the one that minimizes the cost is chosen and the node is updated.

Function OptimizeParent(n)

```
1  Function OptimizeParent (n)
2      p_n ← arg min_{nbh∈N_n} c(x_nbh, x_n) + c_nbh;
3      c_n ← min_{nbh∈N_n} c(x_nbh, x_n) + c_nbh;
```

The core of the DRRT algorithm involves the creation of a branch b by following parent pointers up to the start node. This branch contains every node encountered except for n_new and start. These nodes are excluded because we want to ensure that their position

is not changed. Moreover excluding these two nodes guarantees that every element of the branch has a parent and at least one child.

Function GradientDescent(b)

```
1  Function GradientDescent (b)
2      while gradientDescentTerminationCheck() do
3          foreach b_i ∈ b do
4              if skipGradientDescent(b_i) then
5                  continue;
6              ∇J_V ←getGradient(b_i);
7              t ← 1;
8              while J_V(x_{b_i} − t ∇J_V) > J_V(x_{b_i}) − (t/2)‖∇J_V‖^2 do
9                  t ← βt;
10             temp ← x_{b_i} − t ∇J_V;
11             if isFeasible(x_{p_{b_i}}, temp) AND isFeasible(temp, C_{b_i}) then
12                 x_{b_i} ← temp;
13     foreach b_i ∈ b do
14         if x_{b_i} has changed then
15             Update b_i in nearest neighbor data structure;
```

The location of the nodes for this the branch is then updated using gradient descent, as shown in Function `GradientDescent`. The gradient descent is applied to minimize $J_V = \sum_{n \in \mathcal{T}} c_n$, similarly to the previous section. Line 4 of the algorithm implements a backtracking line search for the step of the gradient descent. It guarantees a decrease of the cost function depending on the norm of the gradient and has be shown to be fast and stable. After the gradient descent is applied, for all the nodes of the branch updated, the location is updated in the nearest neighbor data structure.

Once the branch has been updated, all its elements are added to the queue Q in order to propagate the changes in the rest of the tree. As in the RRT$^{\#}$ algorithm, the queue is ordered by $cost + heuristic$ and the element with the smallest key is treated at each iteration. The element is tried as a parent candidate for all its neighbors and is chosen if it improves the cost. The cost is propagated to all its children, and the modified nodes are added to the queue. The process stops when all nodes in the relevant region $\mathcal{X}_{\text{rel}}$ are processed, that is,

when the smallest element of the queue has a key larger than the goal.

Function PropagateChanges

```
1  Function PropagateChanges()
2      while Q is not empty do
3          el ← Q.popMin();
4          if Key(el) > Best_Cost then
5              break;
6          foreach nbh ∈ N_el do
7              if c_el + c(x_el, x_nbh) < c_nbh then
8                  c_nbh ← c_el + c(x_el, x_nbh);
9                  p_nbh ← el;
10                 Q.update(nbh);
11         foreach c ∈ C_el do
12             c_c ← c_el + c(x_el, x_c);
13             Q.update(c);
```

3.6 Numerical Results

3.6.1 The Hypercube Experiment

In this experiment, we use the HDE and run the algorithms for different dimensions of the problem until the cost is within 3% of the optimal solution. The DRRT algorithm was implemented in the Open Motion Planning Library (OMPL) [81], which provides efficient implementations of the state-of-the-art sampling-based path-planning algorithms, and can be easily integrated with many robotic systems and simulation environments. The DRRT algorithm is compared against RRT* and RRT$^{\#}$. For each algorithm, three variants are compared: uniform sampling, rejection sampling and informed sampling.

In each dimension, and for each variant, 10 simulations were run and the mean and standard deviation were computed. These are shown in Figure 3.8. We see here again, a very minimal influence of the sampling variants. In lower dimensions, the probability of sampling good points is high enough that both RRT* and RRT$^{\#}$ find a near optimal solution faster than DRRT. But the time to find a near optimal solution for those two algorithms

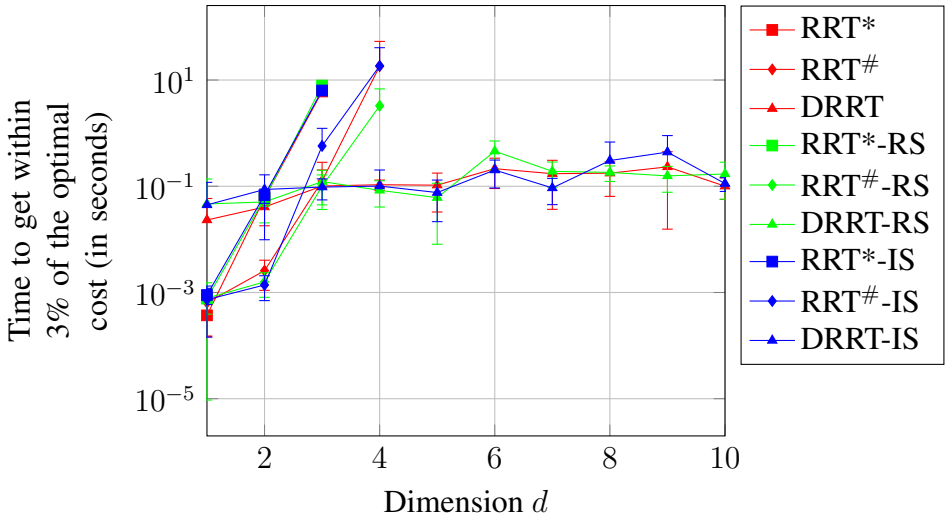


Figure 3.8: Time to converge within 3% of the optimal cost for the hypercube diagonal experiment as a function of the dimension of the problem.

increases extremely fast with the dimension of the problem, whereas for the DRRT algorithm, the time is almost constant. Thus, the DRRT algorithm is faster on this problem by multiple orders of magnitude for any dimension larger than three.

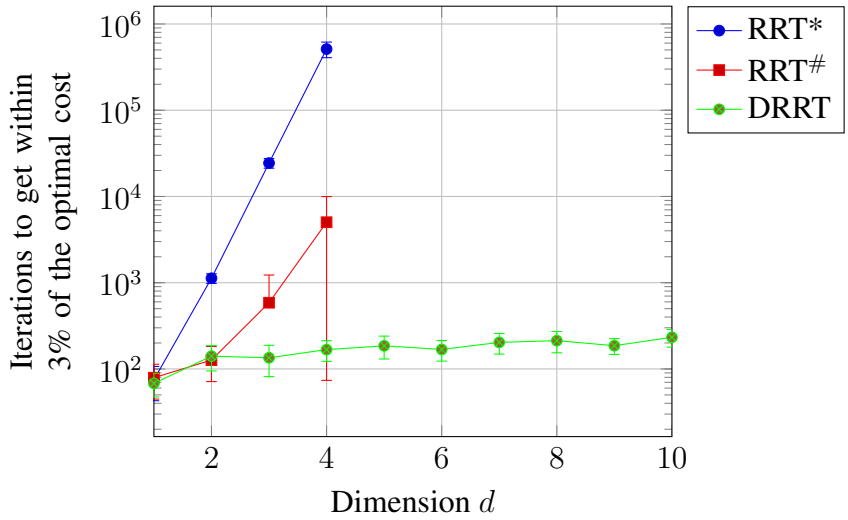


Figure 3.9: Iterations to converge within 3% of the optimal cost for the hypercube diagonal experiment as a function of the dimension of the problem.

Figure 3.9 shows the results of the same experiments, only for the plain variant of each algorithm, in terms of iterations. In dimension two, all algorithms need the same number

of nodes to solve the problem, but as the problem dimensionality grows, this number stays almost constant for the DRRT algorithm, whereas it increases exponentially for the RRT* and the RRT$^{\#}$ algorithms.

3.6.2 6DOF Manipulator

The algorithm was used and benchmarked using the OMPL library [82] against the current state-of-the-art in the V-REP [83] simulation environment on the Mico robotic arm from Kinova Robotics. Figure 3.10 shows the simulation environment. The robot on the right was used for the benchmarking and the trajectories found were used to manipulate the cups. The arm has six degrees of freedom, and the algorithm was benchmarked on three sets of start and goal positions that can be seen in Figure 3.10. For each set, each algorithm was run for 60 seconds and the evolution of the cost over time for each run was reported.

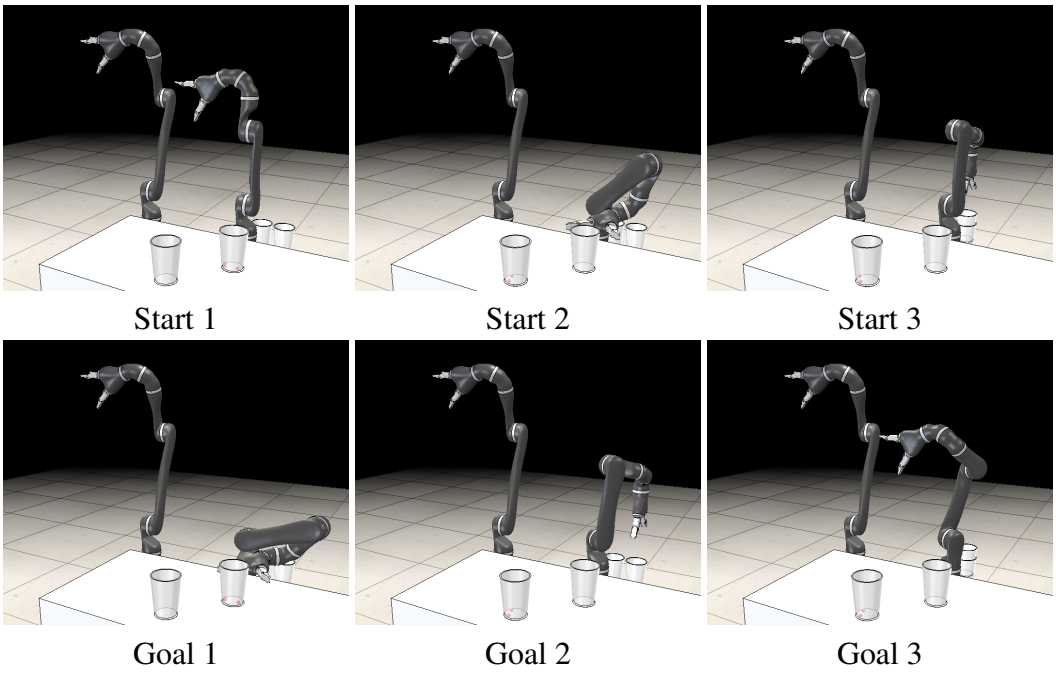
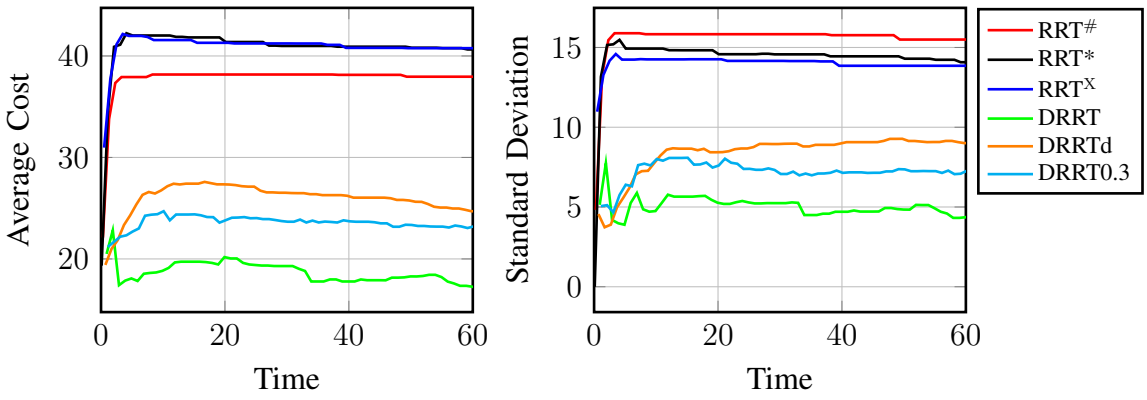


Figure 3.10: Start and goal positions used in V-REP for benchmarking

Figure 3.11 shows the mean of the best cost over all trajectories, as well as the standard deviation. We can see that RRT* finds the largest cost, followed by RRT$^{\#}$ and RRT$^{\mathrm{X}}$

closely together. All three algorithms have a similar standard deviation. Three variants of the DRRT algorithm were tested, the standard algorithm DRRT, a delayed optimization version DRRTd (no optimization is performed until the first solution is found) and a variant where the node optimization is done only 30% of the time, DRRT0.3. The three variants gave significantly better results than the other three algorithms, both in terms of the mean and the standard deviation of solution.



(a) Average Cost

(b) Standard Deviation

Figure 3.11: 6DOF - Best Cost vs Time.

Figure 3.12 shows the cost as a function of the convergence time, the time after the first solution to the goal was found. This Figure allows us to analyze how the algorithms converge once a feasible solution has been found. We can see that for RRT*, RRT$^{\#}$ and RRT^{X} the convergence rate is small, due to the low probability of sampling "good" points in high dimensions. The three DRRT variants show much lower cost for the first solution, and the convergence rate is significantly larger.

The solutions found with DRRT are better, but this improvement comes at a cost. More exploitation of the data means less exploration. Thus the DRRT algorithm is slower to find a first feasible solution. Figure 3.13 shows the percentage of solutions found by those algorithms over time. A total of 100 problems were solved for each pair of start and goal positions. We can see that RRT*, RRT$^{\#}$, RRT^{X} and DRRTd quickly find a feasible solution in all cases, but the DRRT and the DRRT0.3 variants are much slower and do not always

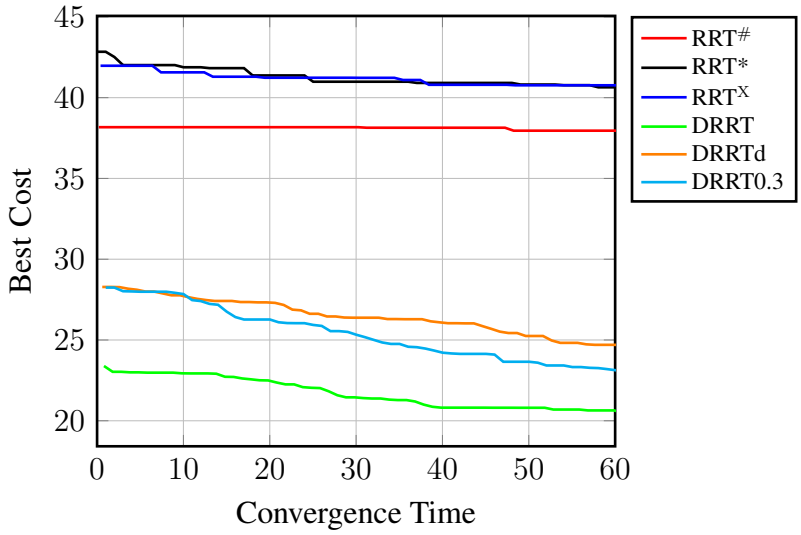


Figure 3.12: 6DOF - Best Cost vs Convergence Time.

find a feasible solution within the allocated 60 seconds.

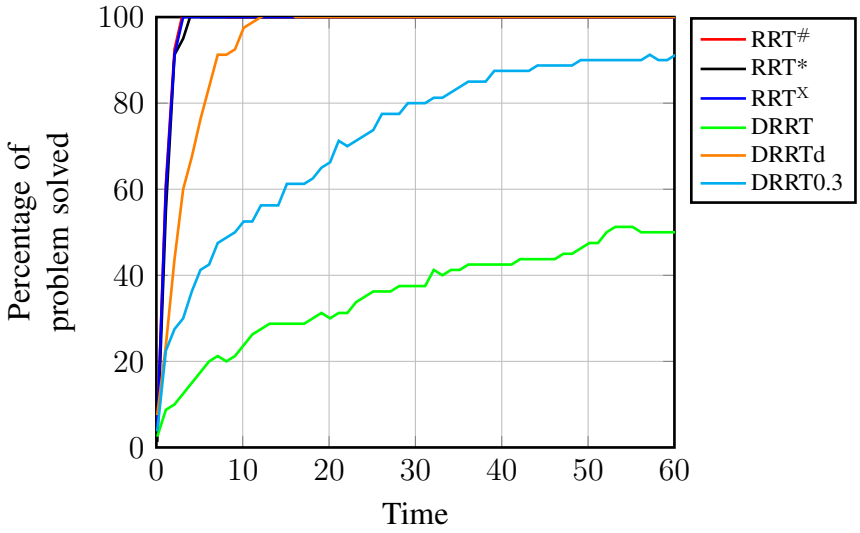


Figure 3.13: 6DOF - Number of solutions found vs Time.

As expected, the ordering of the algorithms by best cost is in the reverse order of the algorithms by time to find the first feasible solution. There is a compromise between the time it takes to find the first solution and the optimality of the solution found. These results suggest that, if time is not a constraint, the DRRT algorithm is the best option as it has the fastest convergence and the lower cost once a solution is found. In the case of limited computational time, a hybrid approach like the DRRTd variant allows to quickly find a

feasible solution before spending computational time in optimizing the sample locations.

3.7 Reducing the Computation of the Gradient Descent

The results of the previous section show that although the DRTT algorithm converges faster, it is slow to find an initial solution. The problem is that too much time is spent exploiting the data, and not enough time is spent exploring the search space. The ratio of exploration versus exploitation is an important parameter to control because different behaviors might be desired depending on the type of environment. Indeed, for an environment with few obstacles, finding an initial solution is easy, so the computational resources can be focused on exploitation to get a faster convergence. On the other hand, if the environment has many obstacles, or some narrow passages, finding an initial solution becomes a hard problem, and the focus should be put on exploration. Moreover, finding an initial solution is not enough to stop exploration, as the optimal trajectory may belong to a homotopy class that has yet to be discovered. This section explores some ideas on how to reduce the time spent doing exploitation in order to be able to increase exploration.

The computation necessary for the gradient descent, as presented for the DRRT algorithm, may be expensive. For each element of the branch of interest, the gradient of the optimization cost needs to be computed, which involves computing the gradient of the planning cost for each child and for the parent node. Additionally, the number of descendants of each child is required. This information can be computed recursively or tracked by the algorithm. However in both cases, it becomes an expensive operation as the tree grows. Finally, when the gradient is computed, all the edges involving the optimized point need to be checked for collision with obstacles.

The following subsections will explore how the computation cost can be reduced by different strategies:

- Applying the gradient descent only on a specific criterion,
- Limiting the number of nodes that are moved by gradient descent,

- Simplifying the computation of the gradient.

3.7.1 When Should the Gradient Descent be Applied?

In order to favor exploration, a criterion can be used to decide whether or not to apply gradient descent at a given iteration. The following criteria have been explored to apply gradient descent if

- A solution to the goal already exists.
- The current branch of interest is part of the best path to the goal.
- There have been N_{OP} iterations since the last optimization was run.

The first criterion allows for time-critical applications to first focus on finding a feasible solution, and to then try to optimize it. This criterion will allow the algorithm to find a solution the fastest, but will result in more vertices in the data structure, which might slow down the optimization later on. The second criterion only optimizes the best path to the goal. It is a good and fast solution for problems with few obstacles, where it is easy to find the optimal homotopy class. In the case of many homotopy classes, focusing on optimizing a single class might slow down the convergence to the optimal solution if it happens to belong to another homotopy class. The third criterion simply runs the optimization less often, providing a simple way to control exploration versus exploitation independently of the state of the algorithm.

Another simple solution is to limit the effort spent on gradient descent by setting termination criteria for the gradient descent algorithm, such as a maximum number of iterations, or a convergence threshold.

Function skipGradientDescentIfNoSolution(b_i)

```
1 Function skipGradientDescentIfNoSolution (b_i)
2     if solutionFound() then
3         return false;
4     else
5         return true;
```

Function skipGradientDescentIfNotOnBestPath(b_i)

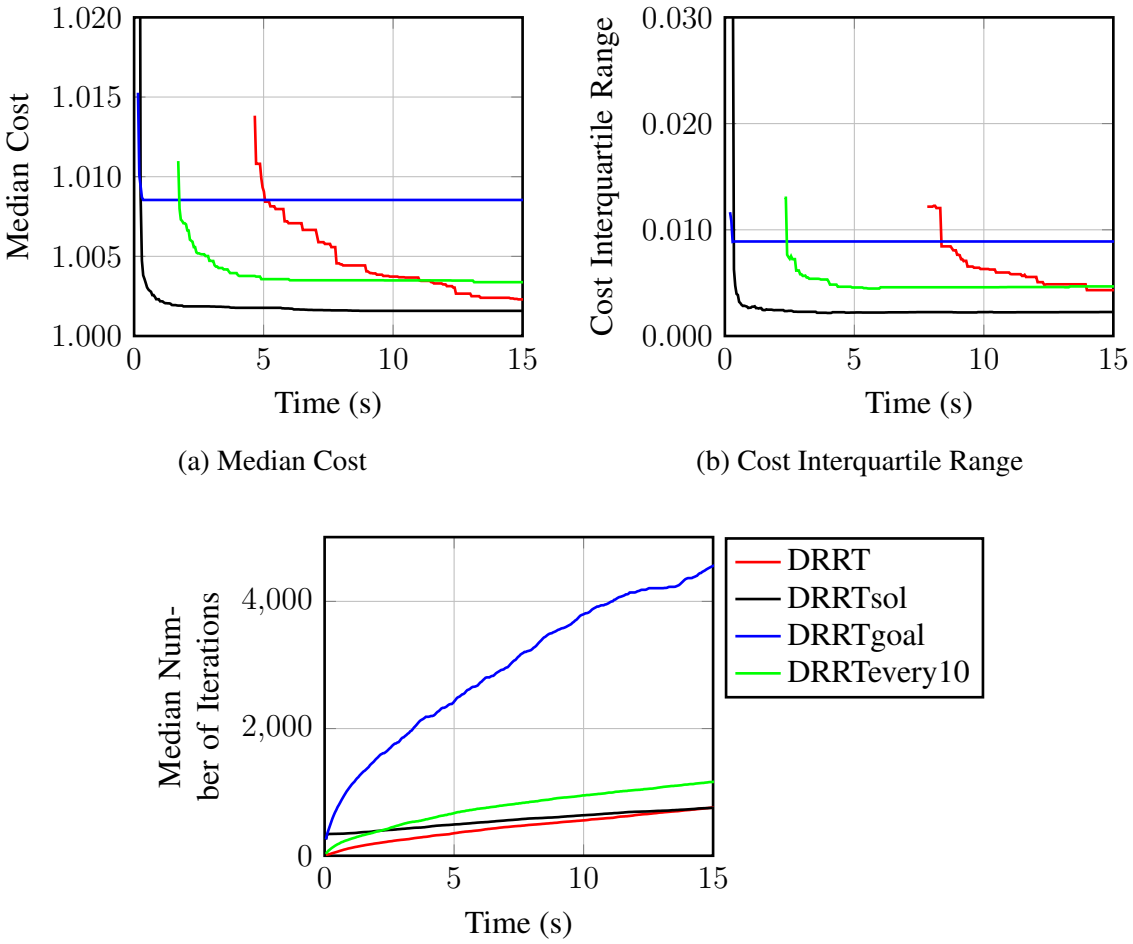
```
1 Function skipGradientDescentIfNotOnBestPath (b_i)
2     if isPartOfBestPath(b_i) then
3         return false;
4     else
5         return true;
```

Function skipGradientDescentApplyEveryN(b_i)

```
1 Function skipGradientDescentApplyEveryN (b_i)
      Parameter: N_OP: Optimization period
2     iterationSkipped ← iterationSkipped + 1;
3     if iterationSkipped > N_OP then
4         iterationSkipped ← 0;
5         return false;
6     else
7         return true;
```

The different gradient descent gating criteria were tested on the HDE in dimension four. It is a very simple example that does not exhibit all the characteristics of each variant, but it gives a good first order evaluation of what typically happens with each variant.

Figure 3.14 depicts the results from a hundred runs of each variant. In terms of median cost (Figure 3.14a), the results of each algorithm appear on this graph if fifty percent of the runs have found a solution. This shows how fast each variant finds a solution. As expected, the variants that do not optimize until a first solution exists (DRRTgoal and DRRTsol) are the fastest ones to find the initial solution. The variant applying the gradient descent every ten iterations (DRRTevery10) is slightly slower to find a solution, but still significantly faster than the regular DRRT algorithm. Due to the low probabilities of sampling a new point on the best branch, the DRRTgoal variant, that only optimizes if the new sample is on the best branch to the goal, converges very slowly. The DRRTsol variant, starting the optimization once a first solution exists, exhibits very fast convergence and a significantly lower cost than any other variants. The DDRTevery10 variant does not seem to provide any advantages for this experiment, since it is not fast to find a first solution and does not show



(a) Median Cost

(b) Cost Interquartile Range

(c) Number of Iterations

Figure 3.14: Gradient Descent Gating.

a very good convergence rate.

Figure 3.14b shows the interquartile range (IQR) of the cost, that is, the difference between the third and the first quartiles. It is a good indicator of how the solutions of the different variants are spread out. The results here are similar to the ones for the cost: DRRTsol performs significantly better than any other variants, and DRRTgoal does not show a significant decrease of the IQR. The regular DRRT and the DRRTevery10 variant end up with a similar IQR at the end of the fifteen second planning time.

Figure 3.14c shows the evolution of the median number of iterations executed by the algorithms over time. DRRTsol and DRRTgoal start with a larger number of iterations since they do not optimize until a first solution exists. Over time, DRRT and DRRTsol

exhibit a similar trend, yet DRRTevery10 executes about fifty percent more iterations, and the DRRTgoal variant performs about four times as many iterations.

This experiment seems to indicate that delaying the optimization until a first solution is found provides the best variant. It is fast to find an initial solution, and then exhibits a very good convergence rate. The DRRTevery10 does not lead to great convergence rate, but it allows exploration of fifty percent more new nodes, which can be beneficial in environments where exploration is of importance.

3.7.2 Reducing the Number of Nodes Optimized

The DRRT algorithm does not differentiate between a leaf of the tree or a node near the root of the tree, but there are major differences between these nodes. A leaf node has most likely never been repositioned by the gradient descent since it only gets updated if it is an internal node of a branch. On the other end, a node near the root of the tree probably has a significant percentage of the entire tree as descendants and thus gets repositioned very often. But for a node near the root, what is practically the same optimization is performed again and again, so the repositioning of this node is most likely yielding very little enhancement of the solution. Some criteria can then be defined in order to reposition nodes only if we expect the operation to yield significant improvement.

One possible criterion would be to simply count the number of times the gradient descent is applied to a given node, and stop running the gradient descent after a threshold is reached. But this approach suffers from the following problem: if a node is already over the threshold and some significant changes happen, like addition or removal of children, the node will not be updated, and significant improvement could be lost.

A different approach is to consider how far up the tree changes might have an impact, that is, we could only update the N_{BD} first ancestors of the current node, where N_{BD} is a parameter to be chosen. The case of $N_{\mathrm{BD}} = \infty$ is the regular formulation of the DRRT algorithm where the entire branch to the root of the tree is updated. Setting $N_{\mathrm{BD}} = 0$, no

gradient descent is applied, so the algorithm becomes similar to RRT$^{\#}$. Another interesting case is $N_{\mathrm{BD}} = 1$, that is, only the parent node of the current node of interest is updated by the gradient descent.

Function skipGradientDescentIfMoreThanNLayersDeep(b_i)

```
1 Function skipGradientDescentIfMoreThanNLayersDeep (b_i)
      Parameter: N_BD: Max depth in the branch to apply gradient descent
2     if i > N_BD then
3         return false;
4     else
5         return true;
```

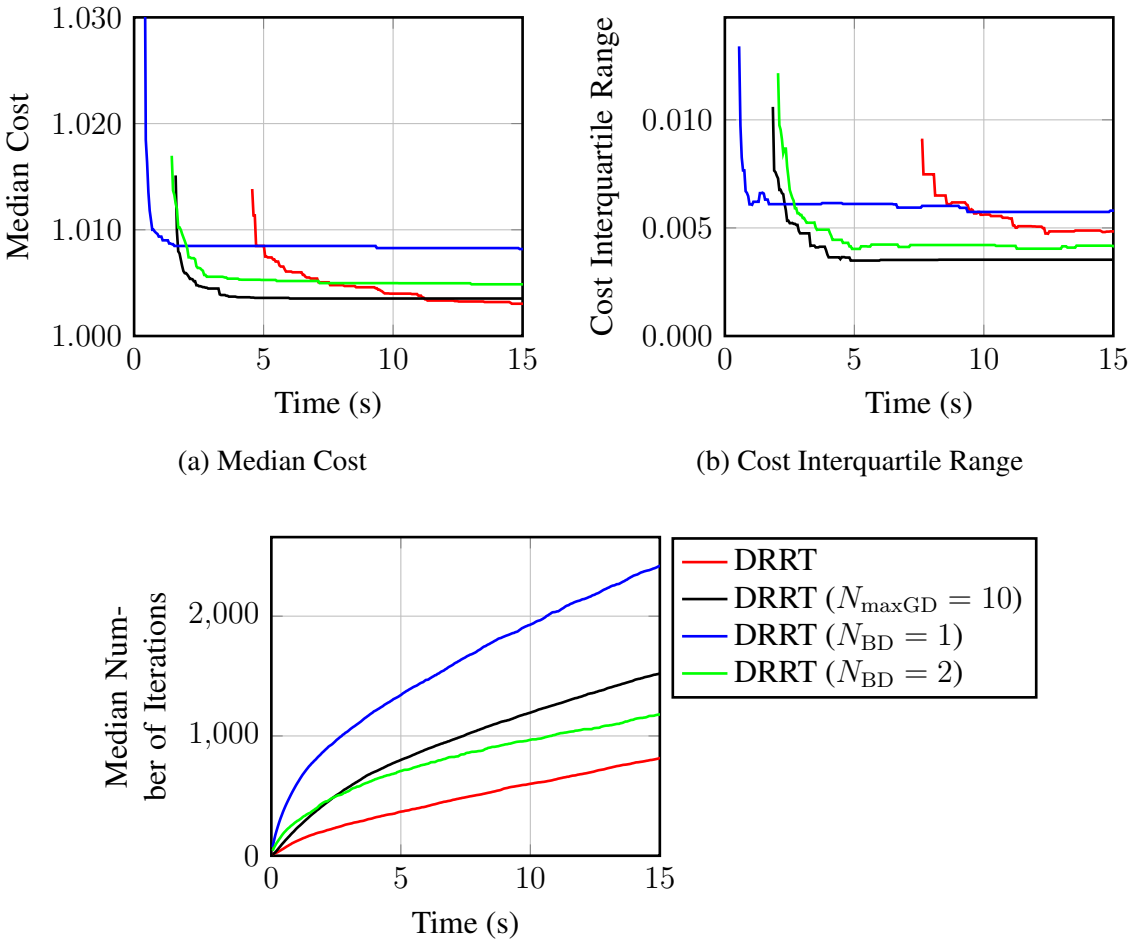
Function skipGradientDescentNoMoreThanNTimes(b_i)

```
1 Function skipGradientDescentNoMoreThanNTimes (b_i)
      Parameter: N_maxGD: maximum number of application of the gradient descent
                 for a given node
2     if numberOfGradientDecentApplied(b_i) > N_maxGD then
3         return false;
4     else
5         return true;
```

All the $skipGradientDescent$ gating functions can be combined with AND and OR gates in order to define more complex behaviors.

Figure 3.15 shows the results of the variants limiting the number of gradient descents. Similarly to the previous section, each algorithm was run one hundred times on the Hypercube Diagonal Experiment in dimension four. As expected, the regular DRRT is the slowest to find an initial solution and it results in the lowest number of iterations over time. However, it ends up being the solution with the lowest median cost at the end of the fifteen second runtime.

Limiting the number of gradient descents applied to a given node ($N_{\mathrm{maxGD}} = 10$) results in a faster first solution, and a good initial convergence, but the cost seems to plateau after a while. Limiting the depth of the branch to which the gradient descent is applied ($N_{\mathrm{BD}} = 2$)



(a) Median Cost

(b) Cost Interquartile Range

(c) Number of Iterations

Figure 3.15: Gradient Descent Gating - Limiting the Number of Gradient Descents

seems to provide a similar behavior, except for a slower iteration rate as the after five seconds.

Stricter limitation on the nodes repositioned ($N_{\text{BD}} = 1$), that is, only a single node repositioned at each iteration, leads to a faster execution of the algorithm. The first solution is found much faster, but with a much larger cost, and the convergence is slow compared to the other variants. The advantage of this variant is that it executes approximately twice as many iterations as the other variants, therefore exploring more of the search space.

3.7.3 From Full Gradient Computation to Simple Fast Approximation

The full gradient descent requires computing the number of descendants of a given node. This operation, albeit simple, requires recursively calling all descendants of the node or keeping track of the number of descendants. In both cases, as the tree grows, the operation becomes expensive.

Although the number of descendants is what is used by the algorithm, the important information needed is the relative weight of each child, that is, the ratio of one child's descendants over the sum of all children descendants. A very coarse assumption would be that the tree is uniform, so each child has the same number of descendants, making them weigh equally. To get a better approximation, one can look one layer farther in the tree and weigh each child by their number of children. Generalizing, the weights of each child can be approximated by looking N_{DD}-layer deep in the tree. This can be done recursively, by limiting the depth of the recursion to N_{DD}. We will consider two specific cases:

- $N_{\mathrm{DD}} = \infty$, the default behavior of the algorithm, that is the full computation of the number of descendants,
- $N_{\mathrm{DD}} = 1$, uniform weighting of the children.

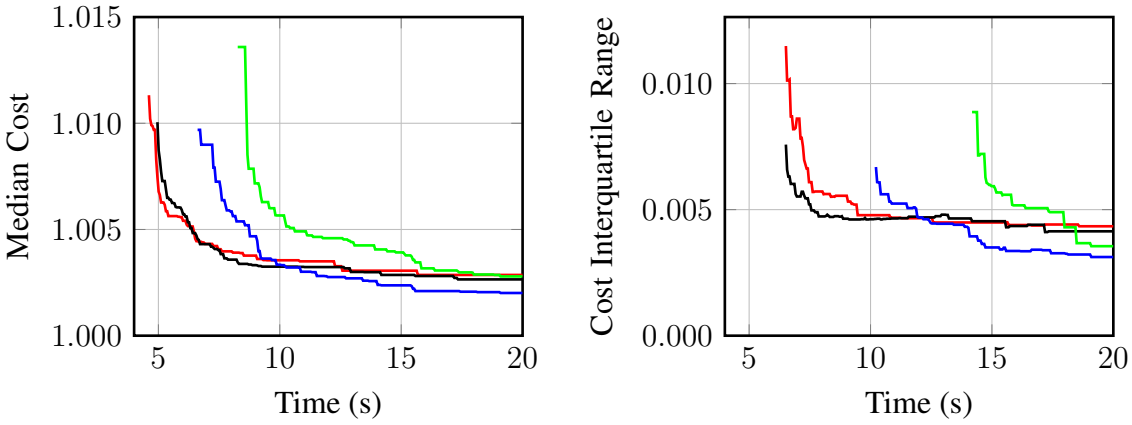
Function getDescendants($node, depth$)

```
1 Function getDescendants (node, depth)
      Parameter: N_DD: Depth used for the number of descendant approximation
2     if depth > N_DD then
3         return 1;
4     count ← 0;
5     foreach c ∈ children(node) do
6         count ← count+ getDescendants (c, depth + 1);
7     return count;
```

Figure 3.16 shows the results of variants using different depths for the computation of the gradient descent. The depth used to compute the weights for the gradient descent does not seem to have a significant effect on convergence, but it does affect how fast iterations

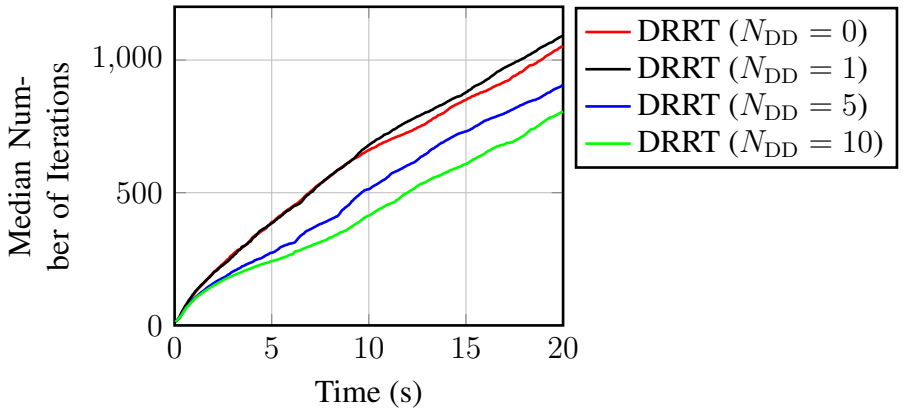
Function getChildrenWeights(*node*)

```
1 Function getChildrenWeights (node)
2     w ← zeros(length(children(node)));
3     foreach c ∈ children(node) do
4         w(i) ← getDescendants (c, 0);
```



(a) Median Cost

(b) Cost Interquartile Range



(c) Number of Iterations

Figure 3.16: Gradient Descent Approximation

are executed. The simplest approximations are the fastest to find a solution, and all variants seem to converge towards the same value. Using $N_{\mathrm{DD}} = 5$, the algorithm results with a slightly lower cost and with a smaller interquartile range. This implies that an optimal value of N_{DD} might exist, but the performance improvements are not significant.

3.8 Variants Results

3.8.1 Kinematic Chain

Problem

The system is similar to a robotic arm in a $2d$ plane. It is composed of n links, connected as a chain and connected to the ground by n joints (see Figure 3.17). The control variables are the angles of the joints $(\theta_1, \theta_2, \ldots, \theta_n)$, making the search space a n-dimensionnal search space. The goal and the obstacles are defined in the (x, y) plane. The problem is to move the end-effector (end of the last link) to given (x, y) position while avoiding collisions between any link and the obstacles.

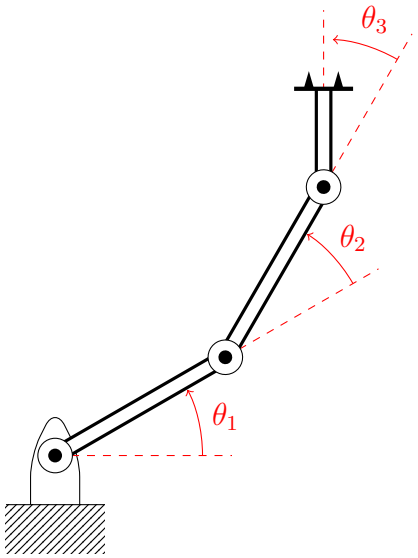


Figure 3.17: Kinematic chain with 3 links.

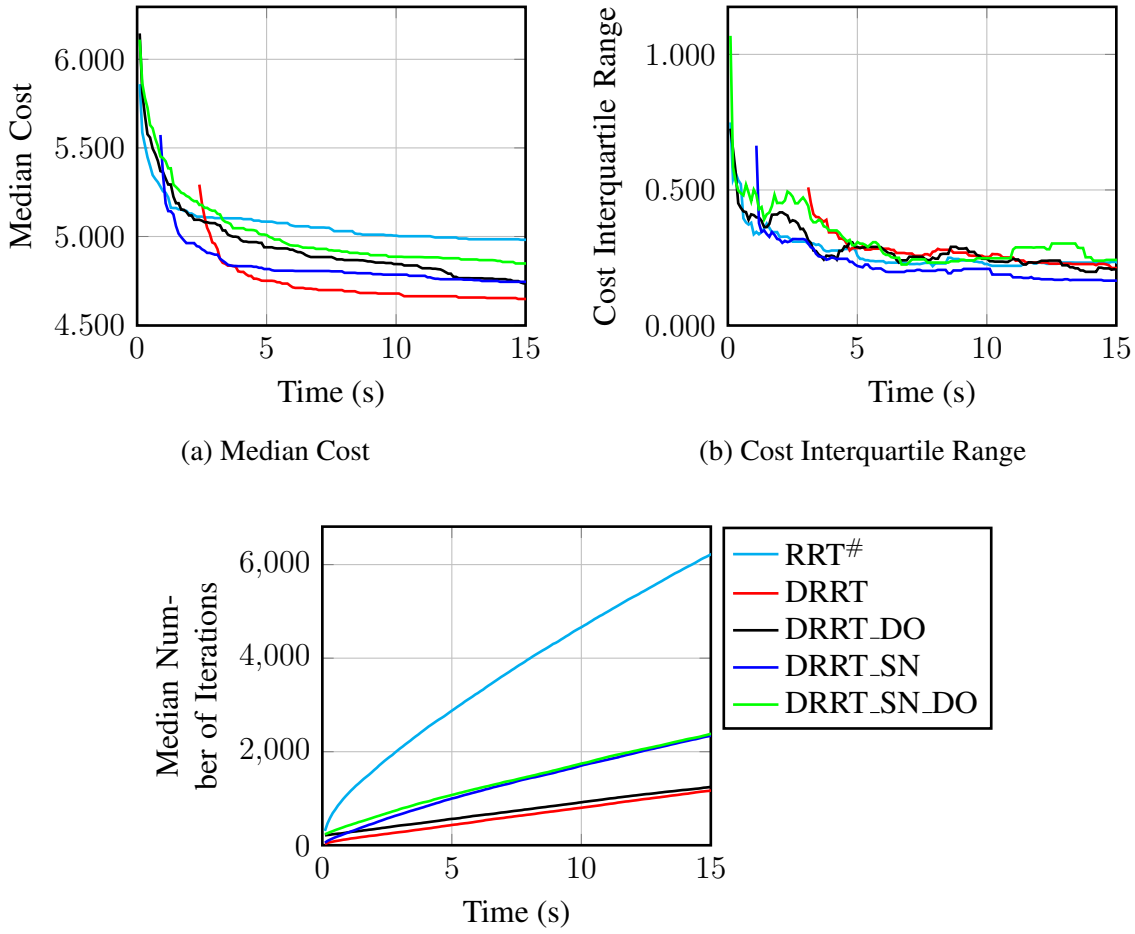
Results

The implementation of the kinematic chain from the Open Motion Library [81] was used. The following variants of the DRRT algorithm are compared:

- suffix **_DO** (Delayed optimization): the gradient descent is only applied once an initial solution is found,

- suffix **_SN** (Single Node optimization): $N_{\text{BD}} = 1$ such that the gradient descent is only applied to the parent of the current node.

In addition, RRT$^{\#}$, the regular DRRT and a version with both variations activated were benchmarked. The gradient weights were all computed using a uniform tree approximation ($N_{\text{DD}} = 1$). Using more precision did not provide significantly better results, and it increased the runtime. Figure 3.18 shows the results of one hundred executions of each algorithm. As one would expect, the regular DRTT is the slowest to find an initial solution



(a) Median Cost

(b) Cost Interquartile Range

(c) Number of Iterations

Figure 3.18: Kinematic Chain Benchmark.

since it does more exploitation of the data gathered. However if time is not a limitation, the DRRT converges faster than the other variants once an initial solution is found, and ends

up with a lower cost.

The delayed optimization (DRRT_DO) provides a much faster solution, similar to RRT$^{\#}$, but convergence is slower. This can be explained by the fact that when a first solution is found, a lot of data has been gathered and needs exploited. The branches are longer and the nodes are farther away from their optimal location, so the gradient descent is applied to more nodes and it takes more time to reach convergence. The single node optimization (DRRT_SN) is slightly slower than the delayed optimization, but still significantly faster the regular DRRT. It provides a fast initial convergence, however over time it does not perform as well as the regular DRRT. Using both variants does not provide very good results, as there is too much data to be optimized after an initial solution is found, and optimizing only one node per iteration makes the convergence slow.

Looking at the number of iterations over time, the RRT$^{\#}$ is by far the fastest, with twice as many iterations as the fastest variant of the DRRT algorithm. This is expected since the gradient descent is an expensive operation, but despite having the fastest iteration rate, the RRT$^{\#}$ has the slowest convergence rate and the highest cost solution. The single node optimization variants both result in a similar number of iterations over time. The DRRT and the DRRT_DO algorithms are the slowest, with a constant offset for the DRRT_DO since no optimization was done until a first solution was found.

3.8.2 Quadcopter Time Optimal Trajectories

Problem

We consider the case of finding time-optimal trajectories for a quadcopter. We make the following assumptions:

- the control inputs are yaw, pitch, roll, thrust (a control innerloop takes care of achieving the desired attitude).
- the velocity is bounded component-wise by user-defined limits.
- the acceleration is bounded component-wise by user-defined limits (maximum pitch

and roll) and some hardware limits (maximum thrust).

Using feedback linearization, we can define the control pseudo-inputs to be F_x, F_y, F_z, M_z, which are the forces applied to the center of gravity of the drone, and the angular momentum around its vertical axis.

Yaw is of little importance for our application, so we decided to leave it free. To do so, we chose to set $M_z = 0$ to limit the control efforts.

The system can then be described by the following equations of motion:

$$m \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix}. \tag{3.28}$$

The acceleration constraints can be moved to the pseudo-control inputs,

$$|F_x| \leq F_{x,\max}, \quad |F_y| \leq F_{y,\max}, \quad |F_z| \leq F_{z,\max}. \tag{3.29}$$

And the velocity constraints are

$$|\dot{x}| \leq V_{x,\max}, \quad |\dot{y}| \leq V_{y,\max}, \quad |\dot{z}| \leq V_{z,\max}. \tag{3.30}$$

Appendix A describes the solver used in order to find the time-optimal trajectories between to states.

Results

On this problem, the DRRT algorithm was compared to the RRT* algorithm and the RRT$^\#$ algorithm. The DRRT algorithm used the delayed optimization, and the gradient descent was applied every fourth iteration. The uniform tree assumption was used to compute the weight of each child in the gradient descent, and the gradient descent was applied to the last two nodes of a branch. Each algorithm was run 50 times, for 40 seconds.

All the algorithms found a solution at the end of the allotted time, and Figure 3.19 shows the distribution of the cost per algorithm at the end of the execution. The median

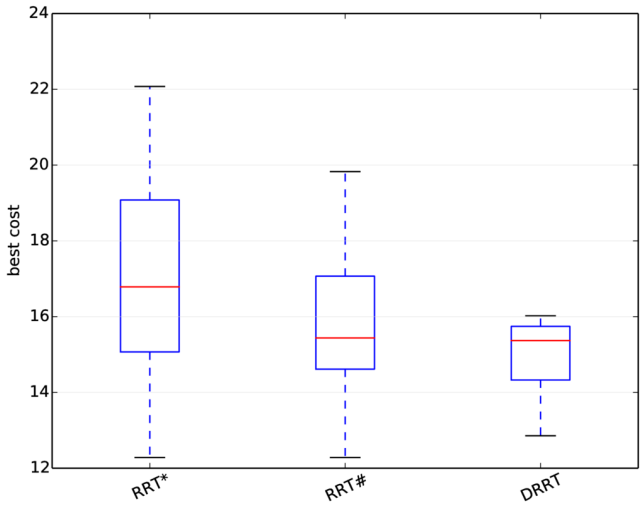


Figure 3.19: Cost distribution per algorithm after 40s

cost found by the DRRT algorithm is similar to the one found by the RRT$^{\#}$ algorithm, but the distribution is much narrower. All the solutions of the DRRT algorithms are below the third quartile of the RRT$^{\#}$ algorithm, and below the median of the RRT* algorithm.

3.9 Discussion

If the DDRT algorithm is used in a similar fashion as the RRT^{X} [56] to take into account moving obstacles, some extra care might need to be taken depending on the problem. As it was seen on Figure 3.5, the gradient descent may create accumulation points at specific locations of the search space. When an obstacle moves over a part of the tree, the RRT^{X} approach is to remove the nodes that do not belong to the free space anymore. If an obstacle moves over an accumulation point, a lot of information could be lost at once. Two approaches could be used to avoid that issue: if the motion of obstacles is known or estimated, this information could be used to move samples such that they stay obstacle-free; a second approach would be to mark a percentage of nodes such that their location is never optimized. This would maintain a portion of samples uniformly distributed over the search

space and guarantee that some information is kept when obstacles move.

Deterministic sampling methods [84] have be shown to work with sampling-based algorithms in [85] and [86]. The use of deterministic sampling removes randomness from the algorithms, making them suitable for safety critical applications; it also provides a slight gain in terms of cost of the solution found. If a deterministic algorithm is desired, the DRRT algorithm can also use deterministic sampling, but it is not clear if it would provide performance enhancement since the DRRT repositions the samples.

3.10 Summary

In this chapter, we introduced a new experiment exhibiting the convergence characteristics of sampling-based planning algorithms as a function of the dimension of the search space of the problem. This experiment was then used to show the convergence limits of sampling-based algorithms as the dimension grows. We introduced a setup to optimize the sample location in order to optimize the cost with a given number of samples. This optimization was integrated in the RRT framework to create the DRRT algorithm, whose results significantly outperform state-of-the-art sampling algorithms. Variants of the DRRT algorithms were also presented with parameters to control how computation is spent in exploitation of the current data versus exploration of the search space. Finally, applications showed the results of the DRRT algorithm and its variants on more complex examples.

CHAPTER 4

CONCLUSION AND DIRECTIONS FOR FUTURE WORK

In this thesis, we discussed the use of multi-scale data structures to accelerate path-planning in large discrete search spaces and the use of classical optimization techniques to increase the convergence rate of sampling-based path-planning algorithms in high-dimensional continuous search spaces. To this end, we proposed a data structure to hierarchically represent the information about the world. This data structure, in the form of a 2^n-tree, provides the ability to get information at different resolutions for a same region of space. The data structure also guarantees the existence of paths between different resolutions. In particular, if no path exists at a coarse resolution, it is proven that no path will exist at a finer resolution.

A novel algorithm, the MSPP algorithm, was introduced to solve the path-planning problem using this new data structure. This new algorithm iteratively solves the path-planning problem on simplified multi-resolution representations of the environment. A backtracking scheme was used to prevent the algorithm from getting stuck, and to ensure that the algorithm finds a solution if one exists.

The MSPP algorithm is proven to be complete. If a solution exists, the algorithm will find it in finite time. If no solution exists, the algorithm will still terminate in finite time and report that no solution exists.

The complexity of the algorithm and the effect of the tuning parameters were analyzed. Numerical experiments validated the theoretical expectations. In particular, the size of the graph used at each iteration is shown to only grow linearly with the depth of the tree data structure, while the size of the actual search space grows exponentially.

Applications to mobile robots exhibit how perception algorithms can use the same data structure. Using the same representation for both perception and path-planning allows for real-time applications. One application also demonstrates the ability of the algorithm to

deal with partially known maps, and to navigate to a goal located in an initially unknown region of space.

After an analysis of the computational bottleneck of the algorithm, further work showed how to reduce the time spent identifying neighboring nodes in the simplified representation of the environment used at each iteration of the algorithm. This enhancement allowed reduction of the runtime by fifty percent for problems with five dimensions.

Since a multi-scale data structure is not always the result of perception algorithms, this thesis also introduced a variant of the algorithm where sampling is used rather than building the multi-scale data structure. This variant also permits the use of the MSPP algorithm on problems where the perception space is different than the planning space (for example, perception in the geometric space and planning in the joint space of a robot). Probabilistic bounds are derived on the probability of not finding a path if one exists. Efficient ordering of slow operations is also done in order to only compute expensive information when required.

An application to the PR2 robot arm showed the use of the MSPP algorithm in the configuration space, and without an a priori multi-scale map. The optimized version of the MSPP algorithm, using the sampling variant, exhibited a runtime improvement by two orders of magnitude.

A second part of the thesis discussed the problem of path-planning on continuous spaces, and the problem of slow convergence of sampling-based path-planning algorithms as the dimension of the search space grows.

To this end, we first developed an experiment exhibiting the convergence properties of sampling-based path-planning algorithms. The results of this experiment concurred the theoretical analysis, and showed that the number of samples required for convergence grows exponentially with the dimension of the problem.

An analysis was then conducted to optimize the estimate of the value function of the path-planning problem for a fixed number of samples. This analysis resulted in an opti-

mization problem that repositions the samples in space. The gradient of the optimization cost was shown to be easily computable, as it only requires local information, namely the local gradient of the trajectory cost with respect to the parent node and the children nodes.

The results of this optimization exhibit a promising behavior. Indeed, for a shortest path length problem with polygonal obstacles, the samples converge towards the vertices of the visibility graph of the environment. Without any assumption about the problem, the optimization recovered that the important points of the environment are the corners of the obstacles.

The sample location optimization was then merged within the framework of RRT algorithms to introduce the DRRT algorithm. The DRRT uses the iterative construction of a tree of paths from the RRT family, and adds a step of optimizing the sample location in order to accelerate convergence. The algorithm significantly outperforms other algorithms on the Hypercube Diagonal Experiment (HDE) by showing a quasi-constant runtime as a function of the dimensionality of the problem versus an exponential increase of the runtime for other algorithms.

Further improvements of the DRRT algorithm were incorporated in order to be able to control the ratio of exploration of the search space versus exploitation of the gathered data. Some variants of the algorithm were presented showing how it is possible to compromise between fast solution and exploration versus fast convergence towards the optimal solution.

4.1 Directions for Future Work

4.1.1 Extension of the MSPP Algorithm to a Generic Cost function

The MSPP algorithm is presented using a cost function that is a linear combination of obstacle probability and distance (see Equation (2.10)). The cost used is defined as a multi-scale cost, which depends on the size of the node considered. Changing the cost function might invalidate Proposition 2 that links the obstacle-free property of a path to its cost, but another test could be used to test if trajectories are obstacle-free. In addition, if ε-obstacles

are excluded from the reduced graph, Proposition 2 may not be necessary, since all paths built on the reduced graph will then be obstacle-free by construction.

Thanks to the backtracking properties of the algorithm, the MSPP can find a solution with any cost function. However, a good multi-scale cost formulation is important such that the path found on the reduced graph provides a good heuristic for the backtracking scheme.

4.1.2 Pruning Nodes in the DRRT Algorithm

As the samples are being repositioned by the DRRT algorithm, they may accumulate at certain locations of the search space. This phenomenon can be seen in Figure 3.5 where the samples accumulate at the corners of the obstacles and at the root of the tree. Exploiting the properties of the problem, minimum length and polygonal obstacles, the path-planning problem could be solved using only 20 samples for this example, resulting in extremely fast solutions.

As the DRRT algorithm runs, those accumulation points appear, but the number of nodes at these points keeps increasing without adding any new information. Instead, they increase the size of the data structure and slow down execution. The idea of pruning the tree is explored in [87] and [55], but the authors use pruning to remove nodes outside the relevant region (note that the implementation of this pruning idea would also be beneficial to the DRRT algorithm).

In the case of accumulation points, a single point (or at least a reduced set of samples) could be used to replace all the samples accumulated. This would result in a reduction of the size of the data structure and the acceleration of the execution of the algorithm.

Choosing the best node representing an accumulation point is a difficult problem. The new samples need to maintain obstacle-free trajectories with the children of all the nodes it replaces. Small cost increases will probably also be seen by these children, which might not be desirable. Also, the computation cost of replacing a set of nodes by a single sample needs to stay small enough to not slow the algorithm. The benefits of reducing the size

of the structure are significant, so exploring this idea could lead to significant performance improvements.

4.1.3 Choice of Optimization Technique in the DRRT Algorithm

The DRRT is presented using a gradient descent algorithm in order to optimize the location of the samples. Strictly speaking, the algorithm uses a coordinate descent algorithm, since the nodes are optimized one after the other. Reference [88] analyzes the properties of coordinate descent algorithms. One risk of coordinate descent algorithms are cycles, as shown in [88], which could lead the DRRT algorithm to spend computational time without any convergence benefit. The gradient solution provides a generic solution that works for any problem or cost function. Our implementation, in the OMPL library, can also compute the gradients numerically, making it a Quasi-Newton method, such that no extra work is needed to set up the DRRT algorithm as the planner for a problem in OMPL. Keeping the algorithm generic provides great flexibility but does not use the specificity of the problem being solved.

The coordinate descent could be parallelized in order to accelerate its execution, or to apply it to more nodes without increasing the runtime. Reference [88] describes two possible parallel implementations, a synchronous and an asynchronous one, and analyzes the expected convergence properties of the each algorithm.

If the cost function used is twice differentiable, then second-order methods can also be exploited to get faster convergence and better numerical stability.

For the case of the Manhattan distance, the minimization Problem 3.22 can be solved solved much faster than by using a gradient descent. The problem can be decomposed into multiple linear problems constraints by linear bounds. These linear problems have many specific optimization methods ([89], [90], [91]) that are much more efficient than a simple gradient descent.

If energy is used as a cost function, the cost function usually takes a quadratic form,

in which case the optimization Problem 3.22 would also have a quadratic form, so the new location of the samples can be computed directly in closed form rather than using a numerical optimization technique.

Appendices

APPENDIX A
BOUNDED ACCELERATION, BOUNDED VELOCITY, TIME-OPTIMAL TRAJECTORIES

The quadcopter application presented in Section 3.8.2 requires the computation of time optimal trajectories for the system. Specifically, given two states X_0 and X_1, representing position and velocity of the quadcopter in three dimensions, the problem is to find the fastest feasible trajectories connecting those two states. Constraints are added to the system, due to hardware and user limitations, and are represented here as component-wise bounds on both velocity and acceleration.

The problem is first solved for the case of a single dimension (position and velocity along one axis), where the solution is a direct application of optimal control. In order to find the optimal solution for multiple dimensions, a solver is also needed to get a fixed-time trajectory for a single dimension in order to synchronize the different dimensions. Finally, the multiple dimension solver puts together the information from the solutions on each dimension.

In this Appendix, a solver for the minimum time problem in a single dimension is presented, then a solver for fixed time trajectories in a single dimension, and finally we introduce a solver for the multi-dimensional minimum time problem.

A.1 Single Dimension

In Section 3.8.2, equation (3.28) shows that for a single dimension, the system is a second order integrator:

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v \\ u \end{bmatrix} \tag{A.1}$$

with x the position, v the velocity and u the control input of the system.

The constraints are

$$|v| \leq V_{\max}, \tag{A.2}$$

$$|u| \leq a_{\max}. \tag{A.3}$$

The optimization cost is the final time of the trajectory

$$J = t_f, \tag{A.4}$$

And the boundary conditions are

$$x(0) = x_0, \quad v(0) = v_0, \quad x(t_f) = x_f, \quad v(t_f) = v_f, \tag{A.5}$$

where x_0 and v_0 are the initial position and velocity, and x_f and v_f are the final position and velocity.

Without the constraint on $|v|$, it is easy to show that this is a bang-bang control and the switching function is a second order polynomial, so there is at most two switches.

Numerically, there are four possible quadratic equations. From these quadratic equation results, the non-physical solutions (negative time) are discarded and the minimum-time physical solution is kept. The solutions can be represented as a list of pairs (Δt, u), which represents the duration Δt of a given segment and the acceleration u during this segment. The trajectory can then be reconstructed by integrating the acceleration.

When adding the constraint on $|v|$, the solutions are still of the same form except when the state constraint is reached. The velocity then stays on the constraint, with no acceleration, until it is supposed to get back below the limit. It is a straight-forward computation from the solution without state constraint. Function `oneDimensionOptimalSolver` presents the pseudo-code used to solve this problem.

Function oneDimensionOptimalSolver(x_0, v_0, x_1, v_1)

```
1  Function oneDimensionOptimalSolver (x_0, v_0, x_1, v_1)
2      bestSol ← ∅;
3      ΔX ← x_1 − x_0;
4      ΔV ← v_1 − v_0;
5      for u ∈ {−a_max, a_max} do
6          Δ ← u ΔX + (v_0^2 + v_1^2)/2;
7          if Δ ≥ 0 then
8              for t_2 ∈ {(−v_1+√Δ)/u, (−v_1−√Δ)/u} do
9                  t1 ← t2 + ΔV/u;
10                 if t_1 ≥ 0 AND t_2 ≥ 0 then
11                     if |v_0 + t_1 u| ≤ v_max then
12                         tempSol ← {(t_1, u), (t_2, −u)};
13                         bestSol ← min(bestSol, tempSol);
14                     else
15                         t_1 ← (sign(u) v_max − v_0)/u;
16                         t_3 ← −(v1 − sign(u) v_max)/u;
17                         t_2 ← (ΔX − v_0 t_1 − u t_1^2/2 − sign(u) v_max t_3 + u t_3^2/2)/(sign(u) v_max);
18                         tempSol ← {(t_1, u), (t_2, 0), (t_3, −u)};
19                         bestSol ← min(bestSol, tempSol);
20     return bestSol;
```

A.2 Single Dimension Fixed-Time Solution

The problem here is to go from (x_0, v_0) to (x_1, v_1) in a given time T. To solve the problem with a fixed time, we first compute the shortest time T^* using Function `oneDimension-OptimalSolver`. If the desired time T is smaller than T^*, then no solution exists. If T is large enough, we try to increase the time of the optimal trajectory by adding a pause. To that end, we check whether the optimal trajectory passes through $v = 0$, and if it does, the system can stop for any desired time at $v = 0$. In particular, stopping for the duration $T - T^*$, will generate a trajectory with a time of exactly T.

If the optimal solution does not go through $v = 0$, the only way to slow down the time taken by the trajectory is to initially "brake" for a period of time Δt. Specific values for Δt can be tried, such as $\Delta t = |v_0|/a_{\max}$, which would bring the system to a stop ($v = 0$), so

a pause could be added if the trajectory time is still smaller than T. Otherwise, Δt can be progressively increased, in order to increase the total time of trajectory.

There does exist one discontinuity in the time of the trajectory as a function of Δt. It happens if after decelerating, accelerating to the desired velocity brings the drone to a position farther than the desired one. In this case, the system needs to go backwards before returning forward to reach the desired position and velocity. This discontinuity can make the time of the trajectory jump from a time smaller than the desired T to a time larger than the desired T, in which case, no solution exists.

A.3 Multiple Dimensions

When the problem is solved for multiple dimensions simultaneously, the constraints are for each dimension, so the problem can still be solved individually along each dimension. However, the solutions in each dimension need to take the same amount of time. To achieve this, we first solve the optimal time for each dimension and keep track of the largest time T needed, since the full trajectory can only go as fast as the slowest dimension. Then we try to solve each dimension with a fixed time T. If all dimensions can be solved, we have a solution. Otherwise, we update T to be the next smallest time that can be achieved by the dimensions that did not have a solution in exactly T. The process is repeated until a solution is found. The problem will terminate since as T increases, every single dimension eventually has time to stop at $v = 0$ and therefore, for a large enough T, there will be a solution where each dimension has a trajectory that takes exactly T.

REFERENCES

[1] M. Ono, T. J. Fuchs, A. Steffy, M. Maimone, and J. Yen, "Risk-aware planetary rover operation: Autonomous terrain classification and path planning," in *IEEE Aerospace Conference*, Big Sky, Montana, Mar. 2015, pp. 1–10.

[2] K. M. Hasan, K. J. Reza, *et al.*, "Path planning algorithm development for autonomous vacuum cleaner robots," in *2014 International Conference on Informatics, Electronics & Vision (ICIEV)*, Dhaka, Bangladesh, May 2014, pp. 1–6.

[3] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, "Practical search techniques in path planning for autonomous driving," *AAAI Workshop - Technical Report*, 2008.

[4] *Jet Propulsion Laboratory*, `https://mars.jpl.nasa.gov/msl/`, Accessed: 2018-12-20.

[5] *Waymo*, `https://waymo.com/`, Accessed: 2018-12-21.

[6] *iRobot*, `https://www.irobot.com/`, Accessed: 2018-12-20.

[7] J. J. Kuffner, "Goal-directed navigation for animated characters using real-time path planning and control," in *Modelling and Motion Capture Techniques for Virtual Environments*, Springer, 1998, pp. 171–186.

[8] M. Lau and J. J. Kuffner, "Behavior planning for character animation," in *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Los Angeles, California, Jul. 2005, pp. 271–280.

[9] J. Pettre, J.-P. Laumond, and D. Thalmann, "A navigation graph for real-time crowd animation on multilayered and uneven terrain," in *First International Workshop on Crowd Simulation*, New York, vol. 43, 2005, p. 194.

[10] N. M. Amato and G. Song, "Using motion planning to study protein folding pathways," *Journal of Computational Biology*, vol. 9, no. 2, pp. 149–168, 2002.

[11] J. Cortés, T. Siméon, V. Ruiz de Angulo, D. Guieysse, M. Remaud-Siméon, and V. Tran, "A path planning approach for computing large-amplitude motions of flexible molecules," *Bioinformatics*, vol. 21, pp. 116–125, 2005.

[12] *Through The Looking Glass*, `http://tlundmark.blogspot.com/`, Accessed: 2018-12-21.

[13] *Wikipedia*, `https://www.wikipedia.org/`, Accessed: 2018-12-21.

[14] I.-C. Chang, H.-T. Tai, F.-H. Yeh, D.-L. Hsieh, and S.-H. Chang, "A VANET-based A* route planning algorithm for travelling time-and energy-efficient GPS navigation app," *International Journal of Distributed Sensor Networks*, vol. 9, no. 7, 2013.

[15] F. B. Zhan and C. E. Noon, "Shortest path algorithms: An evaluation using real road networks," *Transportation Science*, vol. 32, no. 1, pp. 65–73, 1998.

[16] H. Gonzalez, J. Han, X. Li, M. Myslinska, and J. P. Sondag, "Adaptive fastest path computation on a road network: A traffic mining approach," in *International Conference on Very Large Data Bases*, 2007, pp. 794–805.

[17] Y. Kuwata, G. A. Fiore, J. Teo, E. Frazzoli, and J. P. How, "Motion planning for urban driving using RRT," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2008, pp. 1681–1686.

[18] M. McNaughton, C. Urmson, J. M. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 2011, pp. 4889–4895.

[19] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, *et al.*, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[20] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, "Path planning for autonomous vehicles in unknown semi-structured environments," *The International Journal of Robotics Research*, vol. 29, no. 5, pp. 485–501, 2010.

[21] R. Bellman, "On the theory of dynamic programming," *National Academy of Sciences*, vol. 38, no. 8, pp. 716–719, 1952.

[22] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[23] M. Buckland, *Programming Game AI by Example*. Jones & Bartlett Learning, 2005.

[24] A. Stentz and I. C. Mellon, "Optimal and Efficient Path Planning for Unknown and Dynamic Environments," *International Journal of Robotics and Automation*, vol. 10, pp. 89–100, 1993.

[25] A. Stentz, "The Focussed D* Algorithm for Real-Time Replanning," in *International Joint Conference on Artificial Intelligence*, 1995, pp. 1652–1659.

[26] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005.

[27] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A," *Artificial Intelligence*, vol. 155, no. 1-2, pp. 93–146, 2004.

[28] M. Likhachev, D. I. Ferguson, G. J. Gordon, A. Stentz, and S. Thrun, "Anytime Dynamic A*: An Anytime, Replanning Algorithm," in *International Conference on Automated Planning and Scheduling (ICAPS)*, Monterey, California, Jun. 2005, pp. 262–271.

[29] J. Van Den Berg, D. Ferguson, and J. Kuffner, "Anytime path planning and replanning in dynamic environments," in *IEEE International Conference on Robotics and Automation*, Orlando, Florida, May 2006, pp. 2366–2371.

[30] S. Aine and M. Likhachev, "Truncated Incremental Search: Faster Replanning by Exploiting Suboptimality," in *AAAI Conference on Artificial Intelligence*, Bellevue, Washington, Jul. 2013.

[31] ——, "Anytime truncated D*: Anytime replanning with truncation," in *Sixth Annual Symposium on Combinatorial Search*, Leavenworth, Washington, Jul. 2013.

[32] D. Ferguson and A. Stentz, "Using interpolation to improve path planning: The Field D* algorithm," *Journal of Field Robotics*, vol. 23, no. 2, pp. 79–101, 2006.

[33] A. Nash, K. Daniel, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," in *AAAI*, vol. 7, 2007, pp. 1177–1183.

[34] A. Nash, S. Koenig, and C. Tovey, "Lazy Theta*: Any-angle path planning and path length analysis in 3D," in *Third Annual Symposium on Combinatorial Search*, Atlanta, Georgia, Jul. 2010.

[35] A. Yahja, A. Stentz, S. Singh, and B. L. Brumitt, "Framed-quadtree path planning for mobile robots operating in sparse environments," in *IEEE International Conference on Robotics and Automation*, vol. 1, 1998, pp. 650–655.

[36] S. Scheggi and S. Misra, "An experimental comparison of path planning techniques applied to micro-sized magnetic agents," in *International Conference on Manipulation, Automation and Robotics at Small Scales (MARSS)*, 2016, pp. 1–6.

[37] T. Lozano-Pérez and M. A. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *Communications of the ACM*, vol. 22, no. 10, pp. 560–570, 1979.

[38] S. Behnke, "Local multiresolution path planning," in *Robot Soccer World Cup*, Springer, 2003, pp. 332–343.

[39] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems," in *ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation*, Anchorage, Alaska, May 2010.

[40] S. Kambhampati and L. S. Davis, "Multiresolution path planning for mobile robots," *IEEE Journal of Robotics and Automation*, vol. 2, no. 3, pp. 135–145, 1986.

[41] D. K. Pai and L.-M. Reissell, "Multiresolution rough terrain motion planning," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 1, pp. 19–33, 1998.

[42] Y. Lu, X. Huo, and P. Tsiotras, "Beamlet-like data processing for accelerated path-planning using multiscale information of the environment," in *49th IEEE Conference on Decision and Control*, Atlanta, Georgia, Dec. 2010, pp. 3808–3813.

[43] J. Pearl, *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co. Inc. Reading MA, 1984.

[44] R. Bohlin, "Path planning in practice; lazy evaluation on a multi-resolution grid," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 1, Maui, Hawaii, Oct. 2001, pp. 49–54.

[45] R. V. Cowlagi and P. Tsiotras, "Hierarchical motion planning with dynamical feasibility guarantees for mobile robotic vehicles," *IEEE Transactions on Robotics*, vol. 28, no. 2, pp. 379–395, 2012.

[46] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[47] L. Janson, E. Schmerling, A. Clark, and M. Pavone, "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions," *The International Journal of Robotics Research*, pp. 667–684, 2015.

[48] G. Sánchez and J.-C. Latombe, "On delaying collision checking in PRM planning: Application to multi-robot coordination," *The International Journal of Robotics Research*, vol. 21, no. 1, pp. 5–26, 2002.

[49] R. Bohlin and L. E. Kavraki, "Path planning using lazy PRM," in *IEEE International Conference on Robotics and Automation*, vol. 1, 2000, pp. 521–528.

[50] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *IEEE International Conference on Robotics and Automation*, vol. 2, San Francisco, California, Apr. 2000, pp. 995–1001.

[51] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[52] M. S. Branicky, M. M. Curtiss, J. A. Levine, and S. B. Morgan, "RRTs for nonlinear, discrete, and hybrid planning and control," in *42nd IEEE Conference on Decision and Control*, vol. 1, Maui, Hawaii, Dec. 2003, pp. 657–663.

[53] Y. Kuwata, J. Teo, G. Fiore, S. Karaman, E. Frazzoli, and J. P. How, "Real-time motion planning with applications to autonomous urban driving," *IEEE Transactions on Control Systems Technology*, vol. 17, no. 5, pp. 1105–1118, Sep. 2009.

[54] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[55] O. Arslan and P. Tsiotras, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," in *IEEE International Conference on Robotics and Automation*, Karlsruhe, Germany, May 2013, pp. 2421–2428.

[56] M. Otte and E. Frazzoli, "RRT X: Real-Time Motion Planning/Replanning for Environments with Unpredictable Obstacles," in *Algorithmic Foundations of Robotics XI*, Springer, 2015, pp. 461–478.

[57] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "BIT*: Batch Informed Trees for Optimal Sampling-based Planning via Dynamic Programming on Implicit Random Geometric Graphs," *CoRR*, vol. abs/1405.5848, 2014.

[58] A. H. Qureshi, K. F. Iqbal, S. M. Qamar, F. Islam, Y. Ayaz, and N. Muhammad, "Potential guided directional-RRT* for accelerated motion planning in cluttered environments," in *IEEE International Conference on Mechatronics and Automation (ICMA)*, Kagawa, Japan, Aug. 2013, pp. 519–524.

[59] I. Ko, B. Kim, and F. C. Park, "Randomized path planning on vector fields," *The International Journal of Robotics Research*, vol. 33, no. 13, pp. 1664–1682, 2014.

[60] A. Perez, R. Platt, G. Konidaris, L. P. Kaelbling, and T. Lozano-Pérez, "LQR-RRT*: Optimal sampling-based motion planning with automatically derived extension heuristics," *IEEE International Conference on Robotics and Automation*, pp. 2537–2542, May 2012.

[61] D. J. Webb and J. van den Berg, "Kinodynamic RRT*: Asymptotically optimal motion planning for robots with linear dynamics," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2013, pp. 5054–5061.

[62] J. Mainprice, E. A. Sisbot, L. Jaillet, J. Cortés, R. Alami, and T. Siméon, "Planning human-aware motions using a sampling-based costmap planner," in *IEEE International Conference on Robotics and Automation*, Shanghai, China, May 2011, pp. 5012–5017.

[63] B. Akgun and M. Stilman, "Sampling heuristics for optimal motion planning in high dimensions," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Francisco, California, Sep. 2011.

[64] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Chicago, Illinois, September 2014, pp. 2997–3004.

[65] J. Lu, Y. Diaz-Mercado, M. Egerstedt, H. Zhou, and S.-N. Chow, "Shortest paths through 3-dimensional cluttered environments," in *IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong, China, May 2014, pp. 6579–6585.

[66] S.-N. Chow, J. Lu, and H.-M. Zhou, "Finding the shortest path by evolving junctions on obstacle boundaries (E-JOB): An initial value ODEs approach," *Applied and Computational Harmonic Analysis*, vol. 35, no. 1, pp. 165–176, 2013.

[67] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *IEEE International Conference on Robotics and Automation*, Kobe, Japan, May 2009, pp. 489–494.

[68] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "Stomp: Stochastic trajectory optimization for motion planning," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 2011, pp. 4569–4574.

[69] O. Castillo, L. Trujillo, and P. Melin, "Multiple objective genetic algorithms for path-planning optimization in autonomous mobile robots," *Soft Computing*, vol. 11, no. 3, pp. 269–279, 2007.

[70] D. P. Garg and M. Kumar, "Optimization techniques applied to multiple manipulators for path planning and torque minimization," *Engineering Applications of Artificial Intelligence*, vol. 15, no. 3-4, pp. 241–252, 2002.

[71] R. V. Cowlagi, "Hierarchical motion planning for autonomous aerial and terrestrial vehicles," PhD thesis, Georgia Institute of Technology - School of Aerospace Engineering, 2011.

[72] A. Kundu, Y. Li, F. Dellaert, F. Li, and J. M. Rehg, "Joint semantic segmentation and 3d reconstruction from monocular video," in *European Conference on Computer Vision*, Springer, Zurich, Switzerland, Sep. 2014, pp. 703–718.

[73] D. Nister, O. Naroditsky, and J. Bergen, "Visual odometry," in *Computer Vision and Pattern Recognition*, Washington, DC, Jun. 2004.

[74] R. Newcombe and A. Davison, "Live dense reconstruction with a single moving camera," in *Computer Vision and Pattern Recognition*, San Francisco, CA, Jun. 2010.

[75] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2005.

[76] C. Wang, N. Komodakis, and N. Paragios, "Markov random field modeling, inference and learning in computer vision and image understanding: A survey," *Computer Vision and Image Understanding*, vol. 117, no. 11, pp. 1610 –1627, 2013.

[77] J. H. Kappes, M. Speth, G. Reinelt, and C. Schnorr, "Towards efficient and exact map-inference for large scale discrete computer vision problems via combinatorial optimization," in *Computer Vision and Pattern Recognition*, Portland, OR, Jun. 2013.

[78] G. Brostow, J. Fauqueur, and R. Cipolla, "Semantic object classes in video: A high-definition ground truth database," *Pattern Recognition Letters*, vol. 30, no. 2, pp. 88–97, 2009.

[79] L. Ladicky, P. Sturgess, C. Russell, S. Sengupta, Y. Bastanlar, W. Clocksin, and P. H. Torr, "Joint Optimisation for Object Class Segmentation and Dense Stereo Reconstruction," in *British Machine Vision Conference*, Aberystwyth, United Kingdom, Aug. 2010.

[80] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, vol. 34, no. 3, pp. 189–206, 2013.

[81] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, Dec. 2012, `http://ompl.kavrakilab.org`.

[82] M. Moll, I. A. Şucan, and L. E. Kavraki, "Benchmarking Motion Planning Algorithms: An Extensible Infrastructure for Analysis and Visualization," *IEEE Robotics & Automation Magazine*, vol. 22, no. 3, pp. 96–102, Sep. 2015.

[83] M. F. Eris Rohmer Surya P. N. Singh, "V-REP: a Versatile and Scalable Robot Simulation Framework," in *The International Conference on Intelligent Robots and Systems*, Tokyo, Japan, November 2013, pp. 1321–1326.

[84] H. Niederreiter, *Random number generation and quasi-Monte Carlo methods*. Siam, 1992, vol. 63.

[85] A. Yershova and S. M. LaValle, "Deterministic sampling methods for spheres and so (3)," in *IEEE International Conference on Robotics and Automation*, vol. 4, New Orleans, LA, May 2004, pp. 3974–3980.

[86] L. Janson, B. Ichter, and M. Pavone, "Deterministic sampling-based motion planning: Optimality, complexity, and performance," *The International Journal of Robotics Research*, vol. 37, no. 1, pp. 46–61, 2018.

[87] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," *arXiv preprint arXiv:1404.2334*, 2014.

[88] S. J. Wright, "Coordinate descent algorithms," *Mathematical Programming*, vol. 151, no. 1, pp. 3–34, 2015.

[89] J. E. Beasley and J. E. Beasley, *Advances in Linear and Integer Programming*. Clarendon Press Oxford, 1996.

[90] C. Roos, T. Terlaky, and J.-P. Vial, *Interior Point Methods for Linear Optimization*. Springer Science & Business Media, 2005.

[91] J. Matousek and B. Gärtner, *Understanding and Using Linear Programming*. Springer Science & Business Media, 2007.